1988

Exploiting Parallelism Across Program Execution: A Unification Technique and Its Analysis

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Aditya P. Mathur
Purdue University, apm@cs.purdue.edu

Report Number:
88-751

EXPLOITING PARALLELISM ACROSS PROGRAM EXECUTION: A UNIFICATION TECHNIQUE AND ITS ANALYSIS

Vernon Rego
Aditya P. Mathur

CSD TR-751
March 1988

Exploiting Parallelism Across Program Execution: A Unification Technique And Its Analysis

Vernon Rego and Aditya P. Mathur
Department of Computer Sciences
Purdue University
West Lafayette, IN 47906

Abstract: This paper describes a new technique for source-source transformation of sequential programs. We show that the transformed programs so generated provide significant speedups over the original program on vector-processors and vector-multiprocessors. We exploit the parallelism that arises when multiple instances of a program are executed on simultaneously available data sets. This is in contrast to the existing approaches that aim at detecting parallelism *within* a program. Analytic and simulation models of our technique clearly indicate the speedups that could be achieved when several data sets are available simultaneously, as is the case in many fields of interest.

Index terms: vector multiprocessors, program unification, multiple data sets, software testing, urn model, order statistic, simulation.

I. INTRODUCTION

In this paper we present and analyze a technique for source-to-source transformation of sequential programs. We claim that the transformed programs so generated can be parallelized more effectively by existing tools such as those reported in [4,13,15,16]. As shown in Fig. 1, a tool based on our technique will transform a source program for input to any one of the above cited parallelization tools or vectorizing compilers.

Several attempts have been made at discovering parallelism in a sequential program for efficient scheduling of computations on vector-processors (e.g. Cray 1S) and vector-multiprocessors (e.g Cray X/MP, Alliant FX/8 and Cedar). All these machines fit a *shared memory parallel processor* model [29] consisting of L homogeneous and autonomous processors interconnected by a network. Each memory module is accessible by all the processors.

DO-loops within a program have been the major targets for parallelization. In [28] techniques for processor assignment to parallel loops are described. *Loop coalescing* has been proposed in [27,29] as a means to restructure several types of nested loops to singly nested loops. The same work also presents simulation results that show the speedups obtained as a result of parallelization when the number of processors is increased. The *Doacross technique* was introduced by Cytron [10] as a mechanism for modeling and scheduling sequential and vector loops

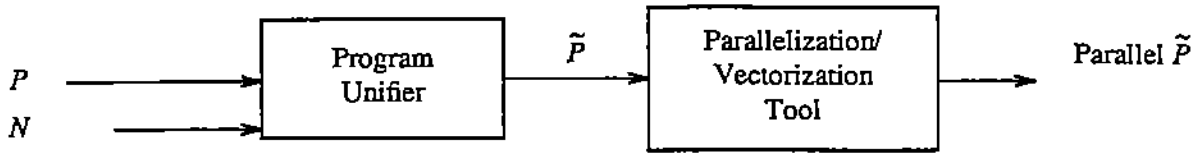


Figure 1. A Tool for Program Unification

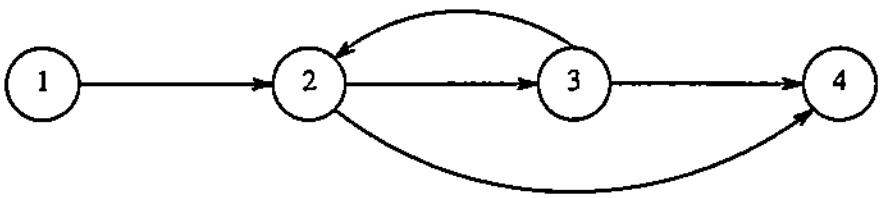


Figure 2. An example of a program graph with four nodes

on multiprocessors. In certain cases, this technique requires insertion of delays during successive execution of loop iterations in order to introduce synchronization. Examples of loops requiring synchronization, and different methods for achieving it, have been exemplified in [22,34].

In [6], it has been pointed out that certain programs perform poorly on a class of machines when they contain:

1. Linear recurrences of order > 1.
2. *IF*-loops performing non-iterative computations.
3. Nonlinear recurrences.
4. *DO*-loops with exits.
5. Short inner loops that depend on outer loops.

The benchmarks that we present in section III show that the transformation technique developed in this paper improves the performance of programs even when they contain statement sequences of the kind cited above.

We note that the benchmarks reported in literature to illustrate the advantages of using the program transformation tools cited above, have not presented any results on programs that employ Monte Carlo simulation techniques. It is a well known fact that such programs perform poorly on vector machines unless special care is taken [31]. Notable attempts have been made however, to develop vector codes for Monte Carlo simulations [8,9]. Techniques described in [9] alter the code structure in order to achieve vectorization. We are not aware of any tool that incorporates these techniques and performs an automatic transformation of the original Monte Carlo code to a vectorizable Monte Carlo code.

The attempts made so far have concentrated on discovering parallelism within *one execution* of a program on given input data. We refer to such an approach as *local optimization*. On the other hand, if a program P is to be executed over several simultaneously available input data sets, the exploitation of parallelism which exists *across* multiple executions of P is what we call *global optimization*. Global optimization may not be of much interest for parallel machines in which individual processors are inherently sequential in nature and capable of operating on multiple instruction streams [14]. The reason why global optimization may not be of much interest in such machines is because instances of P can be scheduled on different processors that work independently. On machines like the Alliant FX/8 or the Cray X/MP, this approach is acceptable if P vectorizes well and can therefore use the resources of a *single* processor efficiently. In general, however, this is not true. As shown in section III, when P does not parallelize, multiple instances of P can be combined together into another program $\bar{P}$ to improve parallelism. In this

paper we refer to $\bar{P}$ as a *unified program.*

The need for simultaneous execution of P on several data sets arises often in a variety of fields. As an example from *software testing*, there are several techniques that require a program to be executed on many test data sets. Regression analysis [7] and *mutation based testing* [1,20] are two such techniques that are computationally intensive. Some experiments done with mutation analysis based testing have shown that global optimization could prove to be of significant utility in many cases. Another area that could benefit from global optimization is *neural modeling.* As an example, computational models of the *cochlea* [3] are often exercised for several combinations of parameter values which include parameters describing the properties of the external sound and non-linear active parameters such as the damping function. These and many other areas serve as the primary motivation for the development and investigation of the transformation presented in this paper.

The remainder of this paper is organized as follows. The next section presents some definitions and terminology for concepts and terms used in this paper. In section III the source-to-source transformation technique, which relies on the global optimization referred to above, is presented with examples. Our transformation produces a program whose dynamic behavior can be analyzed using probability and simulation. An analysis is presented in sections IV and V. Computational results showing the speedup produced by applying our transformation are presented in section VI. We conclude in section VII.

II. DEFINITIONS AND TERMINOLOGY

We shall use P to denote the program to be transformed using the technique described in this paper. We assume that P is to be executed on N simultaneously available data sets denoted by $d_1, d_2, \ldots, d_N$. We refer to P as P_i, an instance of P, when it executes on data set d_i. A *basic block* is a sequence of consecutive statements in which the flow of control enters the first statement and leaves the last statement without the possibility of control leaving at any other statement [2]. Using this definition, P can be transformed [2] into a sequence of K basic blocks denoted by $B_1, B_2, \ldots, B_K$. We will frequently denote these simply as blocks 1,2,3,..... etc., through K. We assume that except for B_K, all basic blocks end with an assignment, an unconditional branch or a conditional branch. Further, a conditional branch at the end of a block B_i is always of the form: if c then *goto label* where c is a scalar logical variable evaluated within block B_i, and *label* is a statement label. We shall denote the label of the first statement of block B_i as s_i.

A. The Program Graph

A given program P operates deterministically on its input. Since we are interested in the behaviour of a program on an arbitrary input, we require some means of defining P's behaviour nondeterministically. For the class of programs that we are interested in studying (e.g., Fortran, Pascal), it can trivially be shown [2] that a given program P with K blocks can be represented by a graph with K nodes. Each node, except for the K^{th} node, has outdegree at most two. The K^{th} node is a terminal node with outdegree zero.

Let G_p be a nondeterministic program graph corresponding to a program P. The graph G_p is obtained by assigning probabilities to the arcs in the deterministic graph of P. As in similar studies [29], we assume that these probabilities are obtained from user supplied expected branching frequencies, or estimated from test data. An example of a program graph is shown in Fig. 2, and an example of a nondeterministic program graph can be seen in Fig. 7(a). Define $\mathbf{R}_m$ to be the set of nodes that can be reached from node m by traversing only a single arc, for $1 \leq m \leq K$. For each m, $1 \leq m < K$, the number of elements in $\mathbf{R}_m$ is at most two, and $\mathbf{R}_K = \Phi$. We use the Greek letter β to denote a branching probability, with the convention that for a given m, $1 \leq m < K$, the probabilities $(1 - \beta_j)$ and β_j are assigned to arcs (m, i) and (m, j), respectively, where $i = \min\{i, j\}$, $j = \max\{i, j\}$, for $\mathbf{R}_m = \{i, j\}$. In case $|\mathbf{R}_m| = 1$ or $|\mathbf{R}_m| = 0$, no confusion arises because the branching probability is either 1 or 0, respectively. Without loss of generality, we assume that block K has outdegree zero, and there is at least one path from block 1 to block K in the program graph.

The time to execute an entity (i.e., a block or a program) x will be denoted by $t(x)$. The program obtained by transforming P using our technique, is denoted by $\bar{P}$. The speedup obtained by concurrent execution of $\bar{P}$ over all the N data sets, against executing P serially over these data sets (i.e. first executing P on d_1, then executing it on d_2 and so on) is denoted by γ, and defined as:

$$\gamma = \frac{\sum_{i=1}^{N} t(P_i)}{t(\bar{P})} \tag{2.1}$$

A block B_i in P gets transformed to block $\bar{B}_i$ in $\bar{P}$. One execution of $\bar{B}_i$ corresponds to N serial executions of B_i. We define the *block speedup coefficient* for block B_i, denoted by α_i, as

$$\alpha_{i,N} = \frac{t(\bar{B}_i)}{(N * t(B_i))} \tag{2.2}$$

and note that $\alpha_{i,N}$ is a typically a decreasing function of N. For vector-multiprocessors, it is a well known fact that $\alpha_{i,N} < 1$ for several types of blocks. We will use α_N to denote the value of the block speedup coefficient averaged over all program blocks of P.

Example: *(Scalar computation)*

For a block in P, containing only one scalar computation $x = y * \sin(z)$, the α values as a function of N are plotted in Fig. 3. † For increasing N, a decrease in the value of α implies that a single execution of the transformed block once is more efficient than an N-step serial execution of the original block.□

The overall speedup γ depends, amongst other factors, on the block speedup coefficients for individual blocks of P [17]. This relationship is made more explicit in section IV.

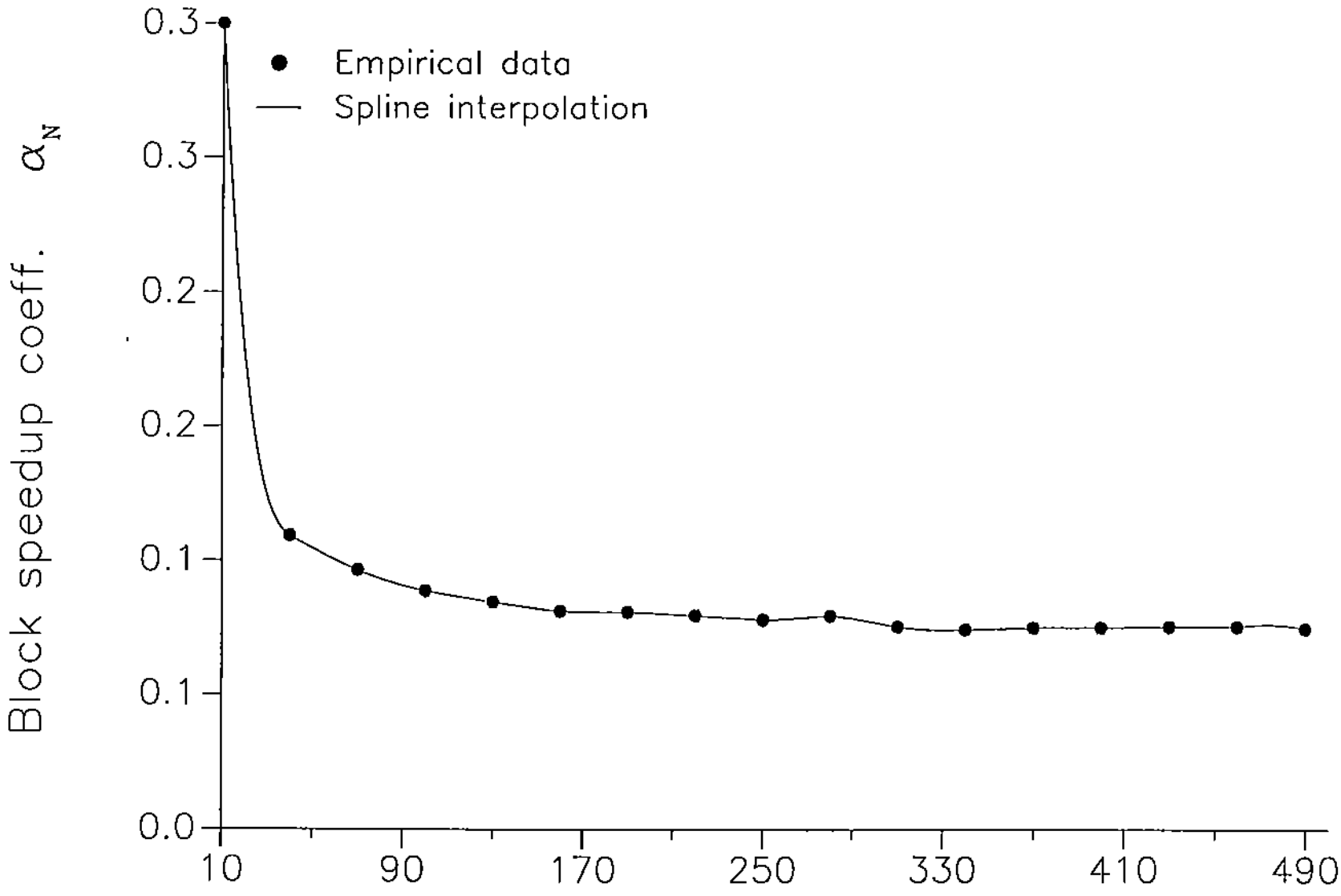
III. THE PROGRAM TRANSFORMATION TECHNIQUE

We shall begin by elaborating the idea of *global* optimization. Suppose that a program P with a flow graph as shown in Fig. 2, is to be executed on three data sets on a uniprocessor vector machine. Further, suppose that the paths followed by P_1, P_2 and P_3 are as shown in Fig. 4(a). Here, blocks which can be executed in parallel are enclosed within a box. Thus, if all three instances of P are to be executed concurrently, block B_1 in each instance can be executed in parallel, followed by block B_2, and then block B_3. At this point, P_1 needs to execute block B_4 and the other two instances of P need to execute block B_2. Assuming that our *block selection* algorithm selects block B_2 as the next one to be executed, P_2 and P_3 can execute this block in parallel. Reasoning along these lines, we can work out the complete execution schedule. Fig. 4(b) exhibits one such schedule.

To show that the above example illustrates a practically viable technique, the remainder of this section is devoted to answering the following questions:

1. How can blocks of different program instances execute in parallel?
2. What mechanisms are needed to manage multiple paths that can arise, as in Fig. 4(a), during the execution of different program instances?
3. What speedup, if any, can be obtained as a result of concurrent execution of multiple instances of P as shown in Fig. 4(a) ?

† All benchmarks presented in this paper have been obtained on the Alliant FX/8 series with one computing element.



Empirical α_N values for x = sin(z)

Fig 3.

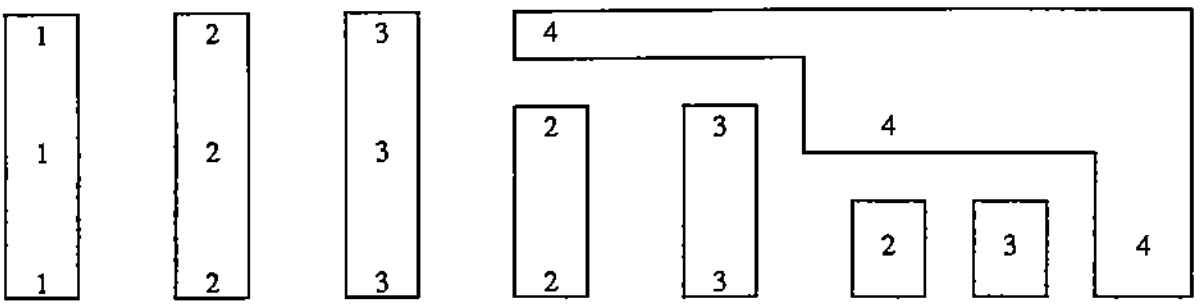


Figure 4(a). Multiple paths in $\tilde{P}$

BLOCK BEING EXECUTED (n)	1	2	3	2	3	2	3	4
Program instances executing the block (n)	1,2,3	1,2,3	1,2,3	2,3	2,3	3	3	1,2,3
Program instances waiting for CPU	none	none	none	1	1	1,2	1,2	none

Figure 4(b). Concurrent execution multiple instance of P

A. Transforming basic blocks

As mentioned earlier, each basic block consists of a sequence of zero or more assignments followed by at most one conditional or an unconditional branch. With this in view, a simple algorithm shown in Fig. 5, named *TRANSFORM*, can be used to transform all the program blocks. Steps 1 and 4 of *TRANSFORM* are to be executed only once for a given P, while the remaining steps are to be executed once for each block. *TRANSFORM* performs a source transformation on P to generate a new source program denoted by $\bar{P}$.

Algorithm TRANSFORM

1. For each variable of dimension D in P, introduce a variable of the same type and of dimension $(D + 1)$ in $\bar{P}$.
2. Each assignment of the form $v = e$ in P gets replaced by

   $DO\ i = 1, N$
   if $(PV(i))\ v(i) = e'(i)$
   $ENDDO$

   The loop variable i is assumed to be a variable not in P. PV is an N element logical vector. In the above transformation, e' has been obtained from expression e using the following rules:

   (a) Constants, function and procedure references remain unchanged.

   (b) A variable referred to as $z(j_1, j_2, \ldots, j_l)$ is replaced by $z(j_1, j_2, \ldots, j_l, i)$. For $l = 0$, this implies that a scalar reference transforms to a vector reference with i as the subscript.

3. An unconditional branch at the end of a basic block remains unchanged. In case a block ends with a conditional branch, it is replaced by the following call: *call outstep (cond, tblock, fblock, nblock)* followed by a branch to a sequence of statements beginning with the statement labeled κ.
4. For each block j, to which there is a branch from itself or some other block, add the following conditional branch: if $(nblock = j)$ goto s_j. The first of these branches should be labeled κ.

Figure 5: Algorithm to transform P to $\bar{P}$

Step 1 of *TRANSFORM* merges the data areas of all instances of P into one data area in $\bar{P}$. Step 2 transforms the assignment statements in a basic block to make the parallel execution of statements across multiple instances of P possible. The transformed statement is guarded by the *partition vector*, which we denote by PV. The i^{th} element of this N element partition vector is *true* if P_i is executing the block containing the guarded statement. After the execution of each basic block, a decision regarding the next block to be executed is needed. A call to routine *outstep* is intended for this purpose. More details of this routine appear in section III-B. The last step of *TRANSFORM* adds a sequence of statements to force the control to branch to the block selected by *outstep*.

Example: *(Nonlinear recurrence)*

As mentioned earlier, nonlinear recurrences are generally difficult to handle by existing tools. In this example we consider the non-linear recurrence:

$$\begin{aligned} x_1 &= 1 \\ x_2 &= 3 \\ x_i &= ax_{i-1}^2 + bx_{i-1} * x_{i-2}, \quad i > 2 \end{aligned} \tag{3.1}$$

Fig. 6 shows how this recurrence can be transformed using *TRANSFORM*, and executed concurrently on N different input data sets each consisting of a set of values of M, a and b. The transformed recurrence was executed on the Alliant FX/8 and its timings compared with the total time to execute the original recurrence for all the N data sets. The resulting speedups are plotted in Figure 6(d) for varying N. As shown in this figure, we obtain significant speedup for $N > 10$.

While measuring the speedup, the time to execute *outstep* has been excluded.□

```
xpp = 1.0
xp = 3.0
DO i = 1, M
    x = a * xp * xp + b * xp * xpp
    xpp = xp
    xp = x
    enddo
```

Figure 6(a): The original non-linear recurrence

```
    xpp = 1.0
    xp = 3.0
    i = 1
/* Block 1 ends here. */
Q: if (i > M) goto R
/* Block 2 ends here */
        x = a * xp * xp + b * xp * xpp
        xpp = xp
        xp = x
        i = i + 1
        goto Q
/* Block 3 ends here */
R: · · ·
```

Figure 6(b): Original recurrence with basic blocks identified

```
/* Initialize across all instances. */
    DO j = 1,N
        xp(j) = 1.0
        xpp(j) = 3.0
        i(j) = 3
        enddo
/* Find instances of P that have completed execution. */
P: DO j = 1,N
        if (pv(j)) cond(j) = i(j) > m(j)
        enddo
/* Select next block to be executed. */
        call outstep (cond,4,3,nextbl)
        goto S
/* Next iteration of recurrence for active instances. */
T:   DO j = 1,N
        if (pv(j)) then
            x(j) = a(j) * xp(j) *xp(j) + b(j) * xp(j) * xpp(j)
            xpp(j) = xp(j)
            xp(j) = x(j)
            i(j) = i(j) + 1
        endif
        enddo
        goto P
S: if (nextbl = 3) goto T
R: · · ·
```

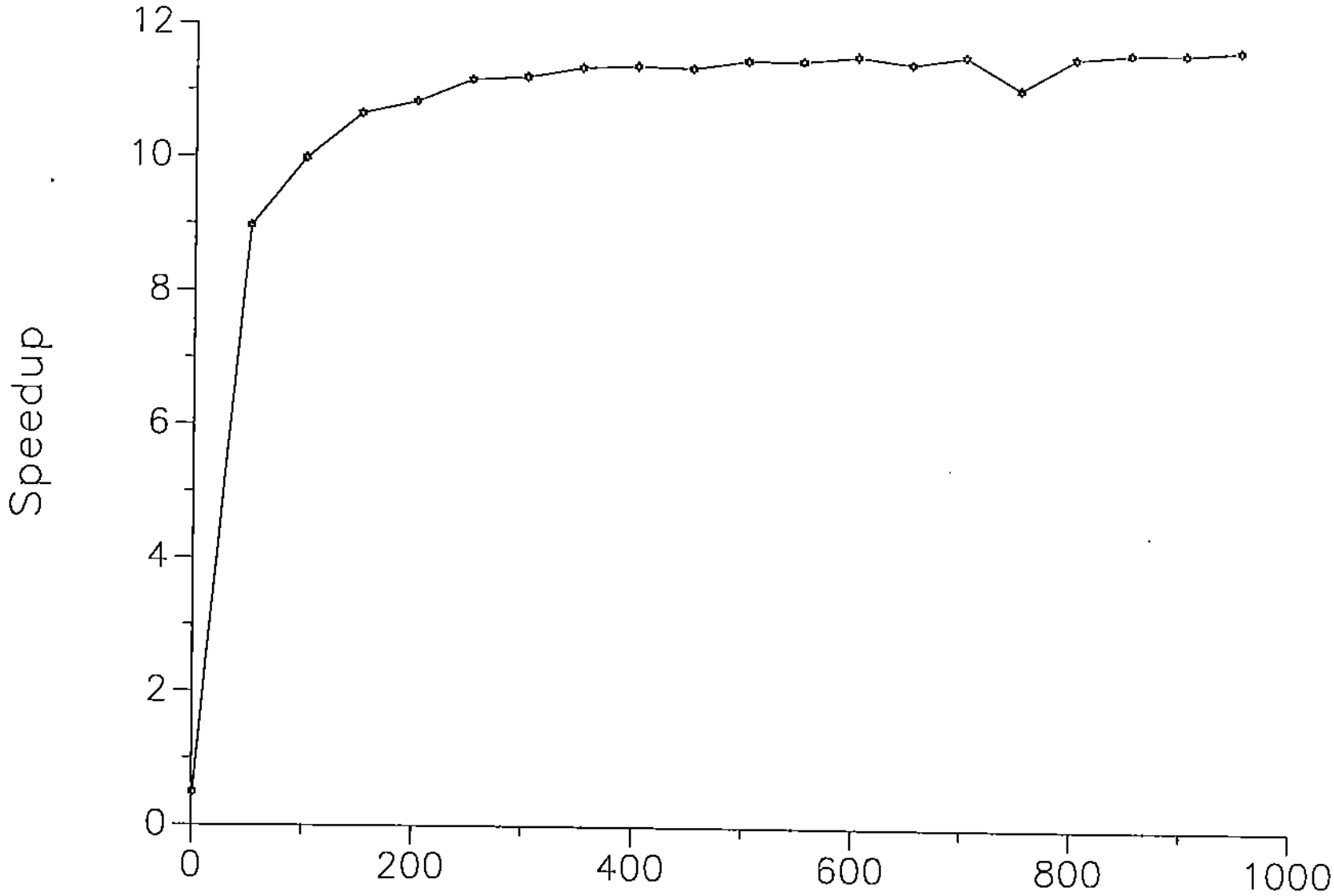
Figure 6(c): Transformed recurrence

The simple example given above indicates the effectiveness of our transformation technique. The same transformation applied on another type of computation results in the speedup curve shown in Fig. 6(e).

B. Managing Multiple Paths

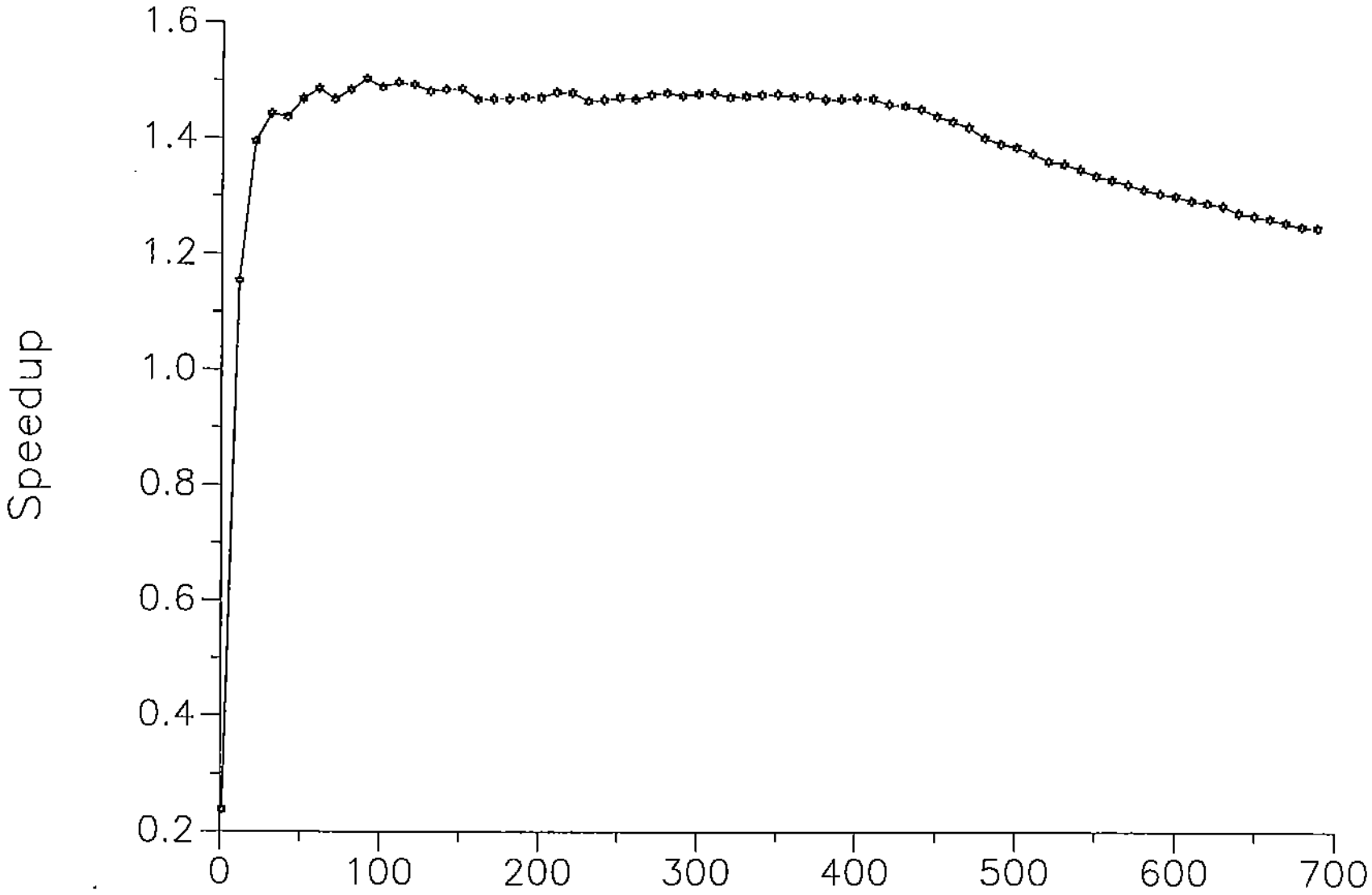
As shown in Fig. 4(a), when the execution of $\tilde{P}$ begins, all instances of P start out by executing block B_1. When the first conditional branch is executed, some of these instances could take one path and the others a different path. On a vector-uniprocessor, only one of these paths could be followed. Program instances that do not follow the selected path, need to wait. This is also true for a vector-multiprocessor if all processors are constrained to execute the same instruction.

Thus, an assignment S in $\tilde{P}$ may not be on the path of one or more instances of P. To insure that the assignment indicated by S is not performed even though the expression on the right side of the assignment may be evaluated, we introduce an N element logical vector termed the *partition vector* and denoted by PV. If the i^{th} instance of P is awaiting execution, or has terminated, $PV(i)$ is false, and otherwise $PV(i)$ is true. The assignment S is guarded by the partition vector as shown in Fig. 6(c).



Nonlinear recurrence speedup (excluding overhead)

Fig. 6(d)



Speedup for dependent inner loop

Fig. 6(e)

All path management related tasks are performed by the *outstep* routine. These tasks are: (1) maintaining a queue of instances of P for each of the K blocks, (2) determining the next block to be executed just after a block has been executed, and (3) updating PV once a block has been selected for execution. The next example illustrates all these tasks when P has a structure as indicated by the flow graph in Fig. 2.

Example: *(Execution sequence)*

One execution sequence for the flow graph of Fig. 2 is shown in Fig. 4(b). Just before block B_1 is executed, PV is set to [*true*, *true*, *true*]. All instances of P move from B_1 to B_2 executing all the intervening statements in parallel. After the execution of block B_2, we assume that *cond* = [*true, true, true*]. At this point *outstep* is invoked. The procedure *outstep* begins by updating the queue of waiting instances for each of the four blocks. After this update, the queues for the four blocks are: B_1 = empty, B_2 = empty, $B_3 = P_1, P_2, P_3$, and B_4 = empty.

outstep now determines the next block to be executed. It uses one of several *block selection policies* to perform this selection. Four of these policies are described in section IV. For this example, we assume that out of all the non-empty queues, *outstep* selects the one with the least index. This leads to the selection of B_3 as the next block for execution. This selection corresponds to the *complete first policy*.

Having selected B_3 for execution, *outstep* resets its queue to empty and sets the partition vector to be [*true*, *true*, *true*]. Now the execution of $\tilde{P}$ begins from block B_3. After the execution of B_3, we have *cond* = [*false, true, true*]. Once again, *outstep* updates the block queues, selects the next block for execution and updates PV to [*true*, *true*, *false*]. This time block B_2 is selected for execution. Thus, only two of the three instances of P execute this block, and P_3 waits its turn at block B_4. This process continues until all instances of P arrive at block B_4. At this point *outstep* selects B_4 for execution, and this block is executed in parallel by all the instances of P. The execution of $\tilde{P}$ terminates at this point.□

At the end of each block that ends in a conditional branch, a call is placed to the *outstep* routine. It is passed three input parameters: *cond, tblock and fblock*, where *cond* is the condition vector evaluated at the end of the block, *tblock* is the block to be executed by instances for which the condition evaluated to true and *fblock* to be executed by instances for which the condition evaluated to false. Using the steps described above, *outstep* determines the next block to be executed and returns this information as the value of the output parameter *nextbl*. This information is then used by $\tilde{P}$ to resume execution from the block indicated by *nextbl*.

IV. A Model for Unified Program Behaviour

In this section we present a class of urn models for studying the behaviour of a unified program on a vector processor. Our goal is to demonstrate, via a model, that executing the unified program $\vec{P}$ on a vector processor yields significant speedup in comparison to a serial execution of program instances P_1 through P_N. A preliminary analysis based on Markov chains was presented in [24]. However, an exact analysis was feasible only for small values of N (number of instances) and K (number of program blocks), typically $N < 10$ and $K < 10$, due to an exponentially growing state space. In this section we develop exact probability models that allow for large values of N and K with little to modest computational effort.

We begin with an intuitive discussion of why speedup is possible through program unification. Following this, we introduce an urn model description of program execution on a vector uniprocessor, based on nondeterministic program graphs (see section II). Using various program graphs, expected block speedup is obtained either explicitly or computationally as a function of N and K.

A. Block Speedup via Program Unification

Given a program P and its nondeterministic graph G_p, we are interested in determining precisely how much is to be gained by merging and executing the N instances $P_1, P_2, ..., P_N$ of P concurrently. While each instance P_j in the unified program $\vec{P}$ makes block transitions according to G_p, the execution paths of P_i and P_j will, in general, be different for $i \neq j$, $1 \leq i, j \leq N$. However, it will frequently be the case that one or more subsets of the set of instances $P_1, ..., P_N$ will require to execute the same block at each execution step. The speedup gain in concurrent program execution is thus directly proportional to the number of program instances converging to execute the same block of the original program P.

Let $t_j \equiv t(B_j)$ denote the time taken by a vector uniprocessor to execute block j in the original program P. During execution, $\vec{P}$ will exhibit maximum processor utilization (and minimum processor wastage due to masking) when all N program instances require to execute the same block simultaneously. For N greater than some threshold integer η, it takes less time to execute the same block, say block j, in $\vec{P}$ than it takes to execute block j serially for all N program instances, i.e. $N\,t_j$. The reason for this is a decrease in execution time for each block of $\vec{P}$, which is a consequence of vectorization.

Assume that when P executes, it visits block j an average of m_j times prior to termination, $1 \leq j \leq K$. The time required to execute P on a single data set is thus $t(P) = \sum_{j=1}^{K} m_j t_j$. Assume now that when $\vec{P}$ executes, it visits block j an average of n_j times prior to termination. The time required to execute $\vec{P}$ is thus $t(\vec{P}) = \sum_{j=1}^{K} n_j t_j N \alpha_{j,N}$, where we recall that $\alpha_{j,N}$ is the block

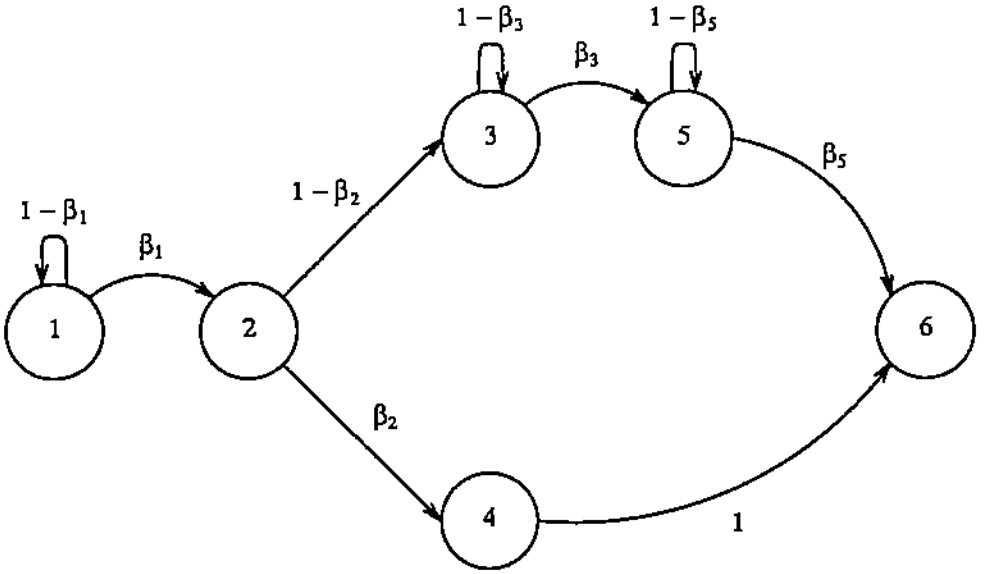


Figure 7(a). Nondeterministic program graph for $K = 6$.

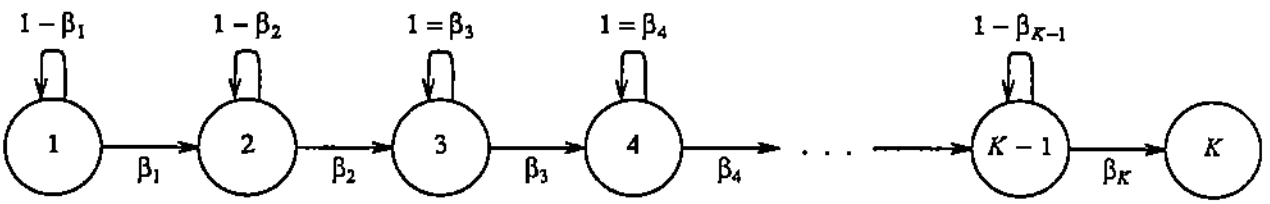


Figure 7(b). A simple program graph.

speedup coefficient (defined in section II) that results from executing N identical blocks simultaneously. Comparing the time to execute P on N data sets serially with the time to execute $\vec{P}$, we obtain the program speedup

$$\gamma = \frac{\sum_{j=1}^{K} m_j t_j}{\sum_{j=1}^{K} n_j t_j \alpha_{j,N}} \tag{4.1}$$

for $N > \eta$.

In the unified program $\vec{P}$, a given block j is visited as often as is required by the program instance P_i which requires to execute this block the most, $1 \le i \le N$. It follows that for each block j, we must have $m_j \le n_j$. Simplifying (4.1), we can bound the speedup from above using

$$\gamma \le \frac{\sum_{j=1}^{K} n_j t_j}{\sum_{j=1}^{K} n_j t_j \alpha_{j,N}}. \tag{4.2}$$

If we make the simplifying assumption that $\alpha_{j,N} = \alpha_N$ is independent of the particular block being considered, (4.2) gives

$$\gamma \le \frac{1}{\alpha_N}, \tag{4.3}$$

and this gives the maximum speedup attainable by $\vec{P}$. On the other hand, the minimum speedup attainable depends on m_j and n_j, for $1 \le j \le N$. To make this explicit, assume that block execution time is the same for all blocks, i.e. $t_j = t$ for $1 \le j \le N$. We obtain

$$\gamma = \frac{\sum_{j=1}^{K} m_j}{\alpha_N \sum_{j=1}^{K} n_j} \tag{4.4}$$

which indicates that m_j and n_j play a major role in attainable speedup. Simplifying to an extreme, assume that $m_j = m$ and $n_j = m$ for each j, $1 \le j \le N$. It follows that speedup is directly proportional to $\frac{m}{n\ \alpha_N}$, implying that speedup is poor when $m < n\ \alpha_N$. This key point led to the development of algorithm *TRANSFORM*, which attempts to (1) minimize α_N by unifying a large number N of programs, and (2) minimize n (or n_j) by scheduling the execution of blocks in $\vec{P}$ in an efficient manner, using the form of G_p to guide this schedule.

B. Urn Model for Execution of P on a Vector Uniprocessor

Assume that we have a set of K urns (say, arranged in increasing order of urn index) representing the K blocks of P. Initially we place a ball in urn 1 and leave it in this urn for a period of length t_1. At the end of this time we pick up the ball from urn 1 and toss it into some urn whose index is in R_1, while consulting the graph G_p in order to determine the probabilities associated with this toss. We proceed in this way, allowing the ball to remain in each urn j that it falls into for a period t_j, before making another toss. When the ball finally reaches urn K, we allow it to remain there for a period t_K and then terminate the procedure.

The above pick and toss procedure is actually a model for the execution of P. We take t_j to be the execution time of block j, $1 \le j \le K$. The time taken from the first toss of the ball into urn 1 until we terminate the tossing procedure is the *execution time* for program P. For an arbitrary graph G_p with one initial and one terminal node, the execution time of P can be shown to be a phase type random variable, i.e., the time to absorption in a finite Markov chain. For a large class of graphs, we can obtain the mean, variance, and indeed even the distribution of this random variable. However, as shown in [24], this is feasible only for small values of N and K.

Example: *(The Execution Time of P)*

Consider the graph shown in Figure 7(a). Define T_j to be the random time taken by the ball from its first visit to urn j until the termination of the procedure (i.e., it gets to urn K and remains there for a time t_K) for $1 \le j \le K$. Let $X(\beta_j)$ be a geometric random variable with parameter β_j, $1 \le j \le K$. That is,

$$Pr[X(\beta_j) = i] = \beta_j(1-\beta_j)^{i-1} \qquad \text{for } i = 1, 2, 3, \cdots \tag{4.5}$$

Merely by examining the graph, we see that

$$T_6 = t_6 \ , \qquad T_5 = T_6 + \sum_{j=1}^{X(\beta_5)} t_5$$

$$T_4 = T_6 + t_4 \ , \qquad T_3 = T_5 + \sum_{j=1}^{X(\beta_3)} t_3 \tag{4.6}$$

Next, defining the mixture

$$T_1 = \begin{cases} T_4 + t_2 & \text{with probability } \beta_2 \\ T_3 + t_2 & \text{with probability } 1-\beta_2 \end{cases} \tag{4.7}$$

we finally obtain

$$T_1 = T_2 + \sum_{j=1}^{X(\beta_1)} t_1 \tag{4.8}$$

as the execution time random variable for P. An application of essentially the same algorithm yields the average execution time of P as

$$E[T_1] = \frac{t_1}{\beta_1} + (t_2 + t_6) + \beta_2 t_4 + (1 - \beta_2)\left\{ \frac{t_5}{\beta_5} + \frac{t_3}{\beta_3} \right\} \tag{4.9}$$

With a little labour, it can be shown that for arbitrary integers m, n, j such that $m \geq 1$, $n \geq 1$, $j \geq 1$, the distribution of P's execution time is given by

$$Pr\ [T_1 = c] = \begin{cases} \beta_2 \beta_1 (1 - \beta_1)^{j-1} & c = jt_1 + t_2 + t_4 + t_6 \\ \beta_1 \beta_3 \beta_5 (1 - \beta_1)^{j-1} (1 - \beta_3)^{n-1} (1 - \beta_5)^{m-1} & c = jt_1 + t_2 + nt_3 + mt_5 + t_6 \end{cases} \tag{4.10}$$

for the specified forms of execution time c. □

C. Urn Model for Execution of $\bar{P}$ on a Vector Uniprocessor

We would like to investigate the behaviour of $\bar{P}$ obtained by merging the N instances P_1, P_2, $\cdots$, P_N of P. Generalizing the urn model outlined above, we now work with N identical balls instead of one. Initially, all N balls are placed in urn 1 and they remain there for a period t_1, representing the parallel execution of the first block in all N programs. At the end of this period all N balls are tossed (again, after consulting G_p) into urns whose indices must be in R_1. The word *toss* is taken to mean that *all* the balls residing in the urn selected for the toss are thrown into target urns *simultaneously*. As before, our goal is to move all N balls into urn K as fast as possible and then terminate the procedure. Unlike the case of a single ball, we see that we will soon reach a stage where the N balls fall into different urns. The question of which urn to select balls from (i.e., which block to execute) for the next toss now naturally arises.

In [24] the authors introduced four block selection policies and studied these via Markov chain methods, obtaining computational solutions for all policies. While the computational solutions clearly demonstrated the speedup given by $\bar{P}$, the Markovian construction led to matrices of high order, making it difficult to determine the best block selection policy. In contrast, our focus is now on (1) obtaining explicit or computational solutions that will enable us to study the behaviour of the unified program $\bar{P}$ for large values of N and K, and (2) determining an optimal block selection policy.

The block selection policies that we choose to consider (from among a host of policies) are:

1. The Complete First Policy (select the nonempty urn with smallest index)

2. The Move Forward Policy (select the nonempty urn with largest index)

4. The Majority Rule Policy (select a nonempty urn with most balls), and

4. The Random Choice Policy (select a nonempty urn randomly)

It is important to note that a uniform feature in all policies is that block K is executed only once, and executed simultaneously by all N program instances. This means that program instances which reach block K ahead of others are required to wait until all program instances arrive at block K in their execution sequences. Finally, a single execution of block K terminates the execution of $\vec{P}$.

V. The Effects of Block Selection Policy

In this section we focus our interest on the behaviour of the different block selection policies, and on obtaining an optimal policy. Consider the simple program graph shown in Figure 7(b). We use this graph to demonstrate just how different results can be when different block selection policies are used. In particular, we compare the speedup given by $\vec{P}$ over the serial execution of $P_1, P_2, \ldots, P_N$ via the Complete First and the Move Forward policies.

A. Overhead Due To Block Selection Policy

While executing the unified program $\vec{P}$, selecting one of a number of candidate blocks for execution on a vector uniprocessor usually involves some computational overhead. It is clear that such overhead depends on both N and K. Henceforth we denote the overhead due to block selection policy as $\delta(N, K)$. This overhead needs to be reckoned with in estimates of program speedup since unified program behaviour will be poor if $\delta(N, K)$ is high relative to block execution times.

B. Serial Execution of $P_1, P_2, \ldots, P_N$

We only need to work with one program (ball) since all programs are identical. Let T_1 be the time taken since a ball is first placed in urn 1 until the termination of the procedure. It is clear (see Fig. 7(b)) that

$$T_1 = t_K + \sum_{j=1}^{K-1} \sum_{i=1}^{X(\beta_j)} t_j \qquad (5.1)$$

where $X(\beta_j)$ is a geometric random variable, with distributional form specified in (4.5). Note that the inner sum in (5.1) represents the sum of a (geometrically distributed) random number of values of t_j. This represents the number of times program control returns to the same block

during program execution, before moving on to the next block. From (5.1) it can be seen that T_1 is a translated binomial random variable. If we use E_P to denote the average execution time of any P_j, $1 \le j \le N$, it follows that

$$E[T_1] = t_K + \sum_{j=1}^{K-1} \frac{t_j}{\beta_j} \tag{5.2}$$

and

$$E_P = N\ E[T_1]\ . \tag{5.3}$$

C. Execution of $\vec{P}$ Under Move Forward Policy

Consider how the Move Forward policy would behave if we began with N balls in urn 1 (see Fig. 7(b)) After $j \ge 1$ tosses of all N balls, we would eventually arrive at a situation in which some balls fall into urn 1 and some fall into urn 2. Suppose, for a moment, that after the j^{th} toss, r balls from urn 1 fall into urn 2, and $(N - r)$ balls remain in urn 1. The Move Forward policy requires that from this point on we ignore urn 1 and work with urn 2 (the largest index nonempty urn). The time to complete the procedure can thus be recursively split into two components, namely, the time to move r balls from urn 2 to urn K, plus the time to move $(N - r)$ balls from urn 1 to urn K.

Let $L(n,m)$ denote the *average* time required from the time N balls fall into urn m until they reach urn K, $1 \le n \le N$, $1 \le m \le K-1$. When balls reach urn K, they remain there until the number of balls in urn K reaches the total value N. When this happens, the balls are allowed to remain in urn K for a period t_K (representing the execution of block K) and then the procedure terminates. Let $A_m(r, n)$ be the *event* that, starting with n balls in urn m, r balls move to urn $m + 1$ on any given toss. We are interested in the first toss in which urn $m+1$ becomes nonempty, since this triggers tosses from urn $m+1$. Let $t_m' = N\ \alpha_{m,N}\ t_m + \delta(N, k)$ be the time taken to process the balls while in urn m, taking into account the overhead due to the block selection policy.

The above reasoning yields the recurrence

$$L(n, m) = \sum_{r=0}^{n} Pr[A_m(r, n)]\ \{\ t_m' + L(n - r, m) + L(r, m + 1)\ \} \tag{5.4}$$

and with some simplification, for $n > 1$, the equivalent recurrence

$$L(n, m) = t_m' + \sum_{r=0}^{n} Pr[A_m(r, n)] \{L(n-r, m) + L(r, m+1)\}$$

$$= \frac{t_m' + \sum_{r=1}^{n} \binom{n}{r} \beta_m^r (1-\beta_m)^{n-r} \{L(n-r, m) + L(r, m+1)\}}{1-(1-\beta_m)^n} \qquad (5.5)$$

since $A_m(r, n)$ describes the event that r balls out of n are chosen on a given toss, from urn m. For $n = 1$, we use

$$L(n, m) = \frac{t_m'}{\beta_m} + L(n, m+1) \qquad (5.6)$$

For practical purposes, we begin the computation by setting the boundary conditions

$$\begin{aligned} &L(n, K) = t_K' && n = N \\ &L(n, K) = 0 && 1 \le n < N \\ &L(0, m) = 0 && 1 \le m \le K \ . \end{aligned} \qquad (5.7)$$

and then iterate, starting at $L(n, K-1)$ and moving downwards to $L(n, 1)$. For each value of m, $1 \le m \le K-1$, we compute $L(n, m)$, starting at $L(1, m)$ and moving downwards to $L(n, m)$, since these values will be required in successive iterations. In this way, the algorithm moves from the last column of a $N \times K$ matrix to the first column of the matrix, from where we obtain the execution time $L(N, 1)$ for the Move Forward policy. Let $E_{\vec{P}}(MF) = L(N, 1)$ denote the expected running time of $\vec{P}$ under the Move Forward policy.

D. Execution of $\vec{P}$ Under Complete First Policy

In using the Complete First policy, the processor continues to alternate between executions and tosses of balls in urn 1 (referring to Fig. 7(b)), moving program control to block 2 only when all N balls are in urn 2. Since the number of tosses required to move any ball from urn 1 to urn 2 is a variable which is independently geometric with parameter β_1, the number of tosses required to empty out urn 1 is precisely the maximum of N independent and identically distributed geometric random variables.

Let $Y(N, \beta_1)$ be the number of tosses required to move N balls from urn 1 to urn 2. The number of tosses required by any particular ball, say ball i, to get into urn 2 is a geometric random variable $X_i(\beta_1)$, for $1 \le i \le N$. Consequently, it follows that

$$Y(N, \beta_1) = \max \{ X_1(\beta_1), X_2(\beta_1) , \ldots , X_N(\beta_1) \} \tag{5.8}$$

is the maximum order statistic from the geometric distribution. It turns out that the distribution of this order statistic, or indeed even any one of its moments, is not trivially obtainable in explicit form. Fortunately, being interested only in the mean of the statistic, we can develop a recurrence for this. In [30] it is shown that the asymptotic form of this order statistic is dominated by $\log_\nu N$, where $\nu = 1/(1-\beta_m)$, which suggests that the Complete First policy is the most efficient for the class of graphs G_P in which urn m is active for time $Y(N, \beta_m)$ and inactive ever after. This is discussed further in the next subsection.

Let $M(n, m)$ denote the *average* number of tosses required to move n balls from urn m to urn K and then terminate, given that urn m is the least index nonempty urn. Clearly, the Complete First policy will begin to work with urn m and will not relinquish control to urn $m+1$ until urn m is empty. The number of tosses required to empty out urn m, given there are n balls in urn m initially, is precisely $E[Y(n, \beta_m)]$. Borrowing from the idea used to develop a recurrence for the Move Forward policy, we see that

$$\begin{aligned} M(n, m) &= t_m' + \sum_{r=0}^{n} Pr[A_m(r, n)]\, M(n - r, m) + M(n, m+1)[1 - (1 - \beta_m)^n] \\ &= \frac{t_m' + \sum_{r=1}^{n} \binom{n}{r} \beta_m^r (1 - \beta_m)^{n-r}\, M(n - r, m)}{1 - (1 - \beta_m)^n} + M(n, m+1) \end{aligned} \tag{5.9}$$

for each m, $1 \le m \le K-1$, with $M(n, K) = t_K'$. When computing the recurrence, we require to set $M(0, m) = 0$, for $1 \le m \le K$. The average time required to terminate the procedure, given that we start by placing N balls in urn m, $1 \le m \le K - 1$ is given by

$$M(n, m) = E[Y(n, \beta_m)] + M(n, m+1) \tag{5.10}$$

so that $\bar{P}$'s complete execution requires

$$M(N,1) = t_K' + \sum_{j=1}^{K-1} E[Y(N, \beta_j]\, t_j' \tag{5.11}$$

units of time. Let $E_{\bar{P}}(CF) = M(N, 1)$ denote the execution time of $\bar{P}$ under the Complete First policy.

E. On establishing an Optimal Policy

As can be seen from the preceding discussion, obtaining analytic estimates of speedup for the graphs G_p in our examples is possible at the expense of little to modest computation. However, given an arbitrary graph G_p, obtaining an estimate of speedup analytically for an arbitrary block selection policy, is a nontrivial task. In this subsection, we use straightforward reasoning to show that for a large class of graphs G_p, the Complete First block selection policy is optimal.

Let G_p be a stochastic graph with set of nodes $S = \{1, 2, \ldots, K\}$. Corresponding to each stochastic graph G_p there exists a unique probability transition matrix $P[G_p]$. Entry (i,j) of this matrix takes on a probability equal to that assigned to the arc of G_p going from node i to node j, $1 \leq i, j \leq K$. If no such arc exists, the (i,j) entry is zero. Define a stochastic graph G_p to be *nonregressive* if the condition $i \leq j \ \forall \ j \in R_i$ is satisfied for each node i belonging to S. A graph is defined to be *regressive* if it is not nonregressive. The graphs shown in Figures 7(a) and 7(b) are clearly nonregressive. In Figure 7(c) is shown a regressive graph that we will use in the next section.

Lemma 1

G_p is a nonregressive stochastic graph if and only if $P[G_p]$ is an upper triangular stochastic matrix.

Proof: If G_p is nonregressive, then $j \geq i \ \forall \ j \in R_j, i \in S$. Consequently, any entry in $P[G_p]$ with $j < i$ will be zero, and $P[G_p]$ will be upper triangular. If G_p is stochastic, then the probabilities on the outgoing arcs of any node must sum to one. Thus the rows in $P[G_p]$ must sum to one. Conversely, if $P[G_p]$ is upper triangular and stochastic, the reverse argument guarantees that G_p is nonregressive and stochastic.

□

In the following discussion, our goal is to show that for the class of nonregressive graphs G_p, the Complete First policy is optimal. We do this by first showing the statement to be true for any pair of connected nodes in G_p, and next applying this step inductively to the entire graph. Another way to look at this is as follows. When $N = 1$, the matrix $P[G_p]$ is precisely the transition probability matrix of a Markov chain. Given that the chain starts in state 1 (i.e., urn 1), the number of block executions required by P in order to terminate is the time to absorption (i.e., number of transitions before state K is reached) in this Markov chain. Lemma 1 tells us that this probability transition matrix is upper triangular.

In the case of $\tilde{P}$, the Markov chain generalizes [24] to one in which each state describes the contents of the K urns. The starting state is given by the single state in which urn 1 has N balls, and the state just prior to termination is the state in which urn K has N balls. With a little labour,

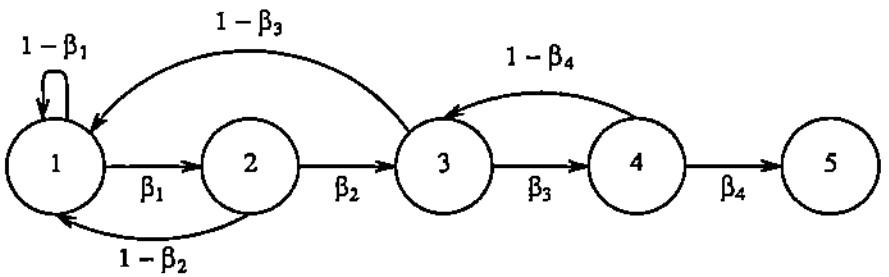


Figure 7(c). A regressive graph.

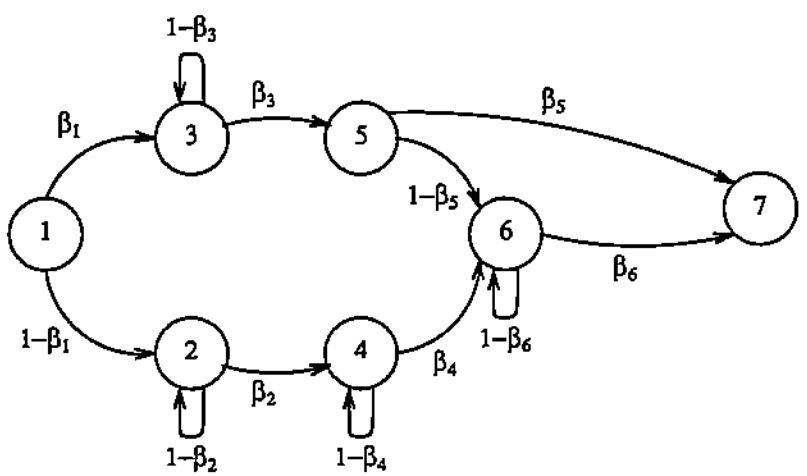


Figure 7(d). A nonregressive graph

it can be shown that no matter what block selection policy is used, the transition probability matrix for the Markov chain describing the behaviour of $\vec{P}$ is always an upper triangular matrix, provided G_P is upper triangular. Further, among the different chains defined by the different block selection policies, the chain given by the Complete First policy is the one whose average number of required block executions, prior to termination, is a minimum. We now prove this without resorting to Markov chains.

Lemma 2

Let G_p be a nonregressive graph with a set of nodes $S = \{1, 2, \ldots, K\}$. For any $j \in S$, let $T(m, j)$ denote the number of tosses required to empty out urn j, given that we begin by placing $m > 0$ balls in urn j. Then $T(m, j)$ is given by

$$T(m, j) = \begin{cases} 1 & \beta_j = 1 \\ 1 & \beta_j < 1,\ j \notin R_j \\ Y(m, \beta_j) & \beta_j < 1,\ j \in R_j \end{cases} \qquad (5.12)$$

where $Y(m, \beta_j) = \max\{X_1(\beta_j), \ldots, X_m(\beta_j)\}$. Additionally $T(m, j)$ is the minimum number of tosses required.

Proof: If $\beta_j = 1$, then all N balls move from urn j into some urn k, in one toss. Since G_p is nonregressive and $\beta_j = 1$, Lemma 1 says that the single entry in row 1 must be in some column k, $k > i$. Thus urn j is emptied forever. If $\beta_j < 1$ and $j \notin S_j$, then all N balls move into a set of urns $R_j = \{k_1, k_2\}$. By Lemma 1, $k_1 > j$, and $k_2 > j$, with $k_1 \neq k_2$. Again, this is done in one toss, and urn j is emptied forever.

If $\beta_j < 1$ and $j \in R_j$, then tossed balls may repeatedly fall back into urn j. The number of tosses required to move any one ball from urn j into some urn k, $k \in R_j$, is a geometric random variable with parameter β_j, i.e., $X(\beta_j)$. The minimum number of tosses required to empty out urn j is thus $Y(m, \beta_j)$, the maximum of m i.i.d. geometric random variables.

□

For nonregressive graphs, it is easy to see that it is best to empty out the nonempty urns in a systematic manner proceeding from the left. The Majority Rule policy tends to distribute balls between urns instead of emptying out urns. The Random Choice policy tends to behave like the latter, but exhibits superior behaviour for nonregressive graphs simply because it sometimes works with the leftmost urn instead of the heaviest urn (i.e., one with most balls). The Move Forward policy is the worst (as can be seen from our example) for such graphs, because it tends to

move individual balls or subsets of balls towards completion, instead of attempting to move all m balls together. We next prove that for this class of graphs, the Complete First policy is the best among all block selection policies.

THEOREM

Let G_p be a nonregressive graph with a set of nodes $S = \{1, 2, \ldots, K\}$. Given an *arbitrary* initial configuration $<n_1, n_2, \ldots, n_k>$ of balls in the K urns, with $\sum_{j=1}^{k} n_j = N$, the Complete First block selection policy is optimal for G_p.

Proof: We proceed by doing induction on the number of urns K. For $K = k$, let M_k be the number of tosses required to terminate the procedure, given that initially there were n_j balls in urn j, for $1 \leq j \leq k$. The theorem is trivially true for $k = 1$, since all n_1 balls exit from urn 1 in one execution step. For $k = 2$, let $<n_1, n_2>$ denote the initial configuration. By Lemma 2, the minimum number of tosses required to terminate the procedure is

$$M_2 = T(n_1, 1) + 1 \tag{5.13}$$

where $T(n_1, 1)$ [see (5.12)] is the minimum number of tosses required to empty out urn 1, given that initially n_1 balls are in urn 1. But M_2 is precisely the number of tosses required by the Complete First policy to terminate the procedure.

Induction Hypothesis: The statement of the theorem is true for $k = m \geq 3$.

Claim: The statement of the theorem is true for $k = m + 1$.

Let $<n_1, n_2, \ldots, n_{m+1}>$ be an arbitrary initial configuration of N balls in $(m + 1)$ urns. According to Lemma 2, the minimum number of tosses required to move n_1 balls from urn 1 into some other set of urns, thus emptying out urn 1 forever, is given by $T(n_1, 1)$ and this is achieved by the Complete First policy. Any other policy would require *at least* $T(n_1, 1)$ tosses.

After $T(n_1, 1)$ tosses, the new configuration of balls would be $<n_1', n_2', n_3', \ldots, n_{m+1}'>$, where $n_1' = 0$. Hence we essentially obtain, within an optimal number of tosses, a new configuration $<n_2', n_3', \ldots, n_{m+1}'>$ of N balls in m urns. An immediate application of the induction hypothesis yields the theorem.

□

It is fairly straightforward to develop an algorithm that computes the exact average execution time for the move forward policy, given an arbitrary nonregressive graph. Unfortunately, the

algorithm can be computationally demanding if the graph contains many nodes j, $j \in S$, satisfying $|R_j| = 2$, $j \in R_j$. These nodes contribute to a large increase in the number of execution sequences. This is best demonstrated with the aid of an example.

Example: *(Execution time for a nonregressive graph)*

Consider the nonregressive graph shown in Figure 7(d). Let $T_j(n)$ denote the amount of time required to empty out urn j, given that it initially contains n balls. Observe that urns 1 and 5 require only a single execution step and this step is independent of the number of balls in these urns when they execute. Urn 7 requires a single execution step, and it does not execute until it contains N balls. Urns 2, 3, 4 and 6 execute a random number of times where this random number is the maximum order statistic from a geometric distribution. Thus we obtain

$$M(N, 1) = T_1(N) + \sum_{i=0}^{N} B_1(i, N)[T_2(N-i) + T_3(i) + T_4(N-i) + T_5(i) + \sum_{j=0}^{i} B_6(j, i)T_6(n-j)] + T_7(N) \tag{5.14}$$

where

$$B_k(j, i) = \binom{i}{j} \beta_k^j (1 - \beta_k)^{i-j} \tag{5.15}$$

for $1 \le k \le K$. A simple algorithm to obtain an approximate value of $M(N, 1)$, and one which is asymptotically exact, is easy to obtain. This algorithm works well for sufficiently large N, say $N > 20$. It is based on the execution time of each urn given the average number of balls in an urn. For an arbitrary nonregressive graph G_p, define a node to be one of three types, i.e., $j \in S$ implies that j belongs to one of the sets **A**, **B**, or **C** where

$$\begin{aligned} \mathbf{A} &= \{ j \mid \beta_j = 1 \} \\ \mathbf{B} &= \{ j \mid \beta_j < 1, \ j \in R_j \} \\ \mathbf{C} &= \{ j \mid \beta_j < 1, \ j \notin R_j \} \end{aligned} \tag{5.16}$$

At any stage during the execution of the Complete First algorithm, if urn k executes (i.e., we toss from urn k), then we agree that out of r balls initially in urn k, $\lfloor \beta_k r \rfloor$ balls go to urn i and $r - \lfloor \beta_k r \rfloor$ balls go to urn j, for $R_k = \{i, j\}$, $k \in C_k$. Asymptotically, this is the exact

average dispersion of balls into urns i and j. In order to compute the average execution time of the Complete First policy for a given G_p, we must determine the average amount of time spent by the procedure with each urn. If the node in G_p corresponding to the urn in execution lies in set A or B, then all the balls in this urn eventually move into a single target urn. However, if the node lies in the set C, then we need to account for the average number of balls that move into each of the two target urns. We do this recursively in the following algorithm. Define a set U to be a set of two-tuples of the form $\{m, n_m\}$, where m is the index of some urn, and n_m is the average number of balls present in the urn. Beginning with the execution of urn 1, we execute urns in increasing order of urn indices, stopping when we finally arrive at urn K.

```
C: global (set);

procedure T(k, n_k);

begin

                    C   ← C ∪ {k, n_k} ;
                    k   ← min_m {{m, n_m} | {m, n_m} ∈ C} ;
                    n   ← Σ_{{k, m} ∈ C} n_k ;

    if( k = K ) then

                    T   ← t_K'

    else

                    begin
                    i   ← min {r | r ∈ R_k} ;
                    C   ← C \ {{k, n_k} | {k, n_k} ∈ C} ;
    Case A:         T   ← t_k' + T(i, n);
    Case B:         T   ← E[Y(n, β_k)] t_k' + T(i, n) ;
    Case C:          begin
                    j   ← max {r | r ∈ R_k} ;
                    n_j ← ⌊ β_k n ⌋ ;
                    n   ← n − n_j ;
                    C   ← C ∪ {j, n_j} ;
                    T   ← t_k' + T(i, n) ;
                    end;
    end;

end.
```

Figure 8: Algorithm for computing $E_F(CF)$

VI. ANALYTIC AND EXPERIMENTAL RESULTS

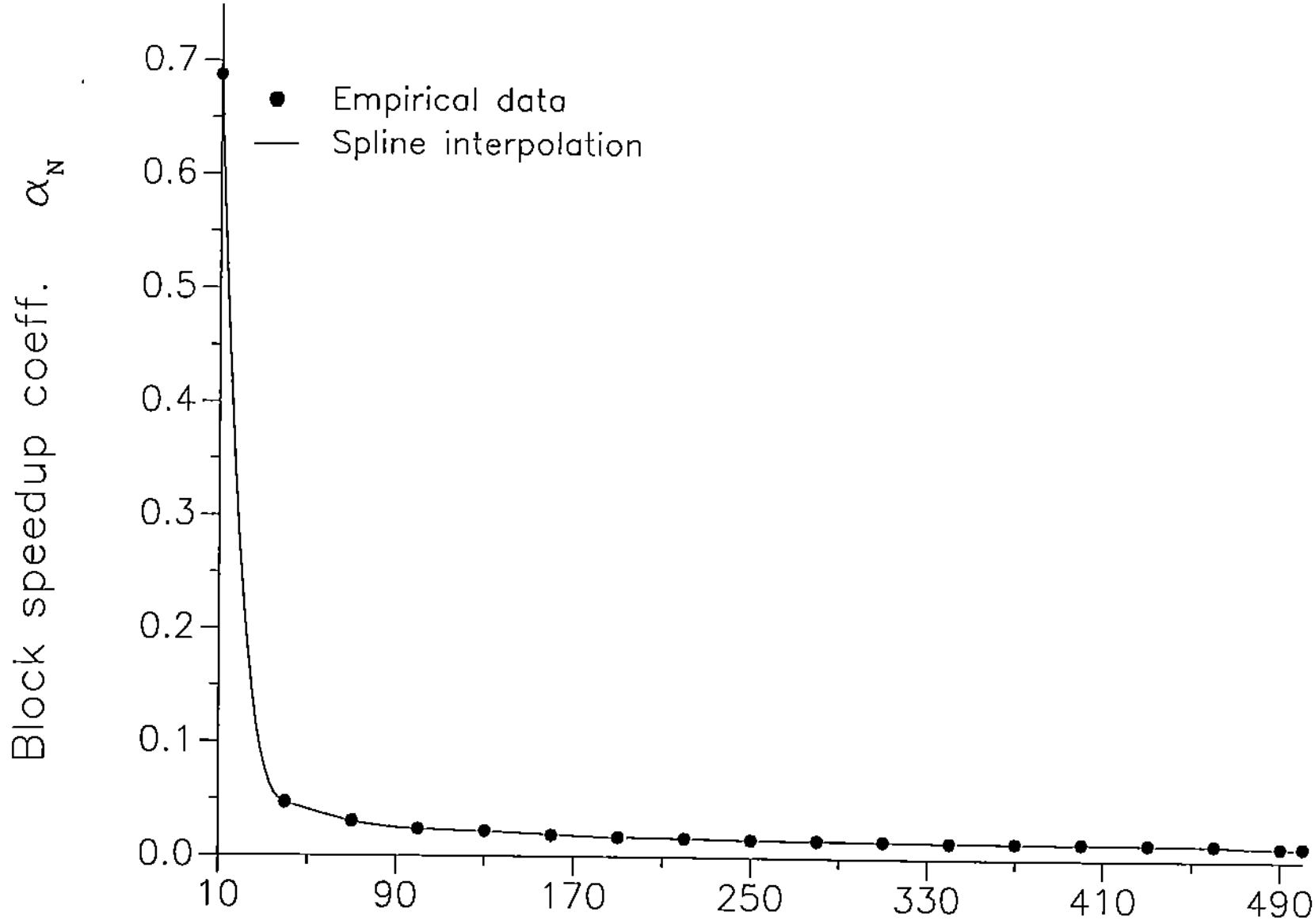
In this section we present graphical results for speedup based on the analysis given in the preceding sections. In all cases, we find that the transformation technique yields significant speedup over serial program execution, for a wide class of program graphs. Clearly, using arbitrary program graphs makes for difficult interpretation problems, since an infinite number of different program graphs can exhibit the same average execution time behaviour. This problem is equivalent to the problem of characterizing absorption times in finite state Markov chains and, in general, is not a well understood problem.

For ease of our own understanding of the average behaviour of *TRANSFORM* for a given block selection policy, we restrict ourselves to examining certain kinds of program graphs, i.e., graphs that possess a certain structure. This enables us to (1) obtain analytic results, (2) establish proofs of optimality, and (3) interpret our computational results in terms of the given graph structure. Additionally, we develop simulation models of the urn and ball tossing procedure to help us study arbitrary program graphs, or understand block scheduling policies that are otherwise difficult to analyze.

The following results all depend on the function shown in Fig. 9 which relates α_N to N. This function was obtained by making block speedup measurements on the Alliant FX/8 over various block types, and then averaging. The continuous line connecting the α values is a spline interpolation, and for the most part is a strictly decreasing function of N. However, for values of N close to 500, the spline falls steeply. As we shall point out later, this artifact results in speedup graphs that tend to climb rapidly for N close to 500.

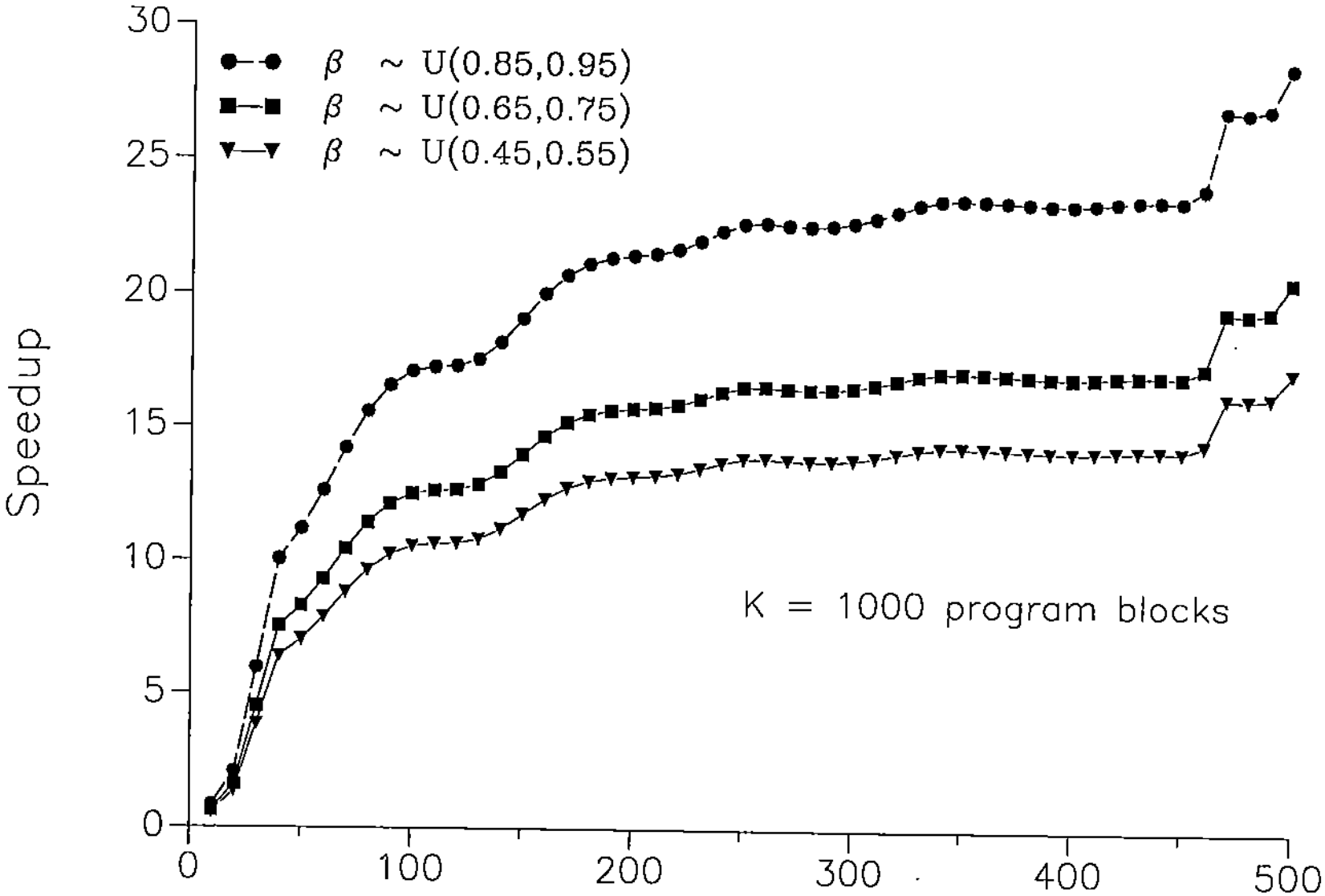
In Figs. 10(a) and 10(b) we display analytic results for the Complete First and Move Forward policies, respectively. These were obtained with the aid of equations (5.3), (5.7) and (5.11), using the graph structure shown in Fig. 7b, with $K = 1000$. The values of β_j, $1 \leq j \leq K$, were selected randomly (i.e., uniformly) from specified intervals, ensuring that these were probabilities. Fig. 10(a) demonstrates that $E_{\bar{P}}(CF)$ can be very small, thereby yielding tremendous speedup. Additionally, we see that speedup is a decreasing function of β. As P tends to iterate more within a block before moving on to the next block, speedup deteriorates due to increasing path conflict between program instances. It is instructive to observe that for this simple structure, the Move Forward policy exhibits negative speedups. The reason for this is the tendency of the Move forward policy to run the fastest instances (i.e., instances that require to execute blocks closest to block K) to completion before executing others, thereby increasing total execution time. It should be clear that for such graphs, $E_{\bar{P}}(CF) < E_{\bar{P}}(MF)$ for all choices of β.

In Figs. 10(c) and 10(d) we use the analytic results to validate the simulation model. This is done for $K = 50$ and β_j chosen randomly from the interval (0.45, 0.55), for the same graph



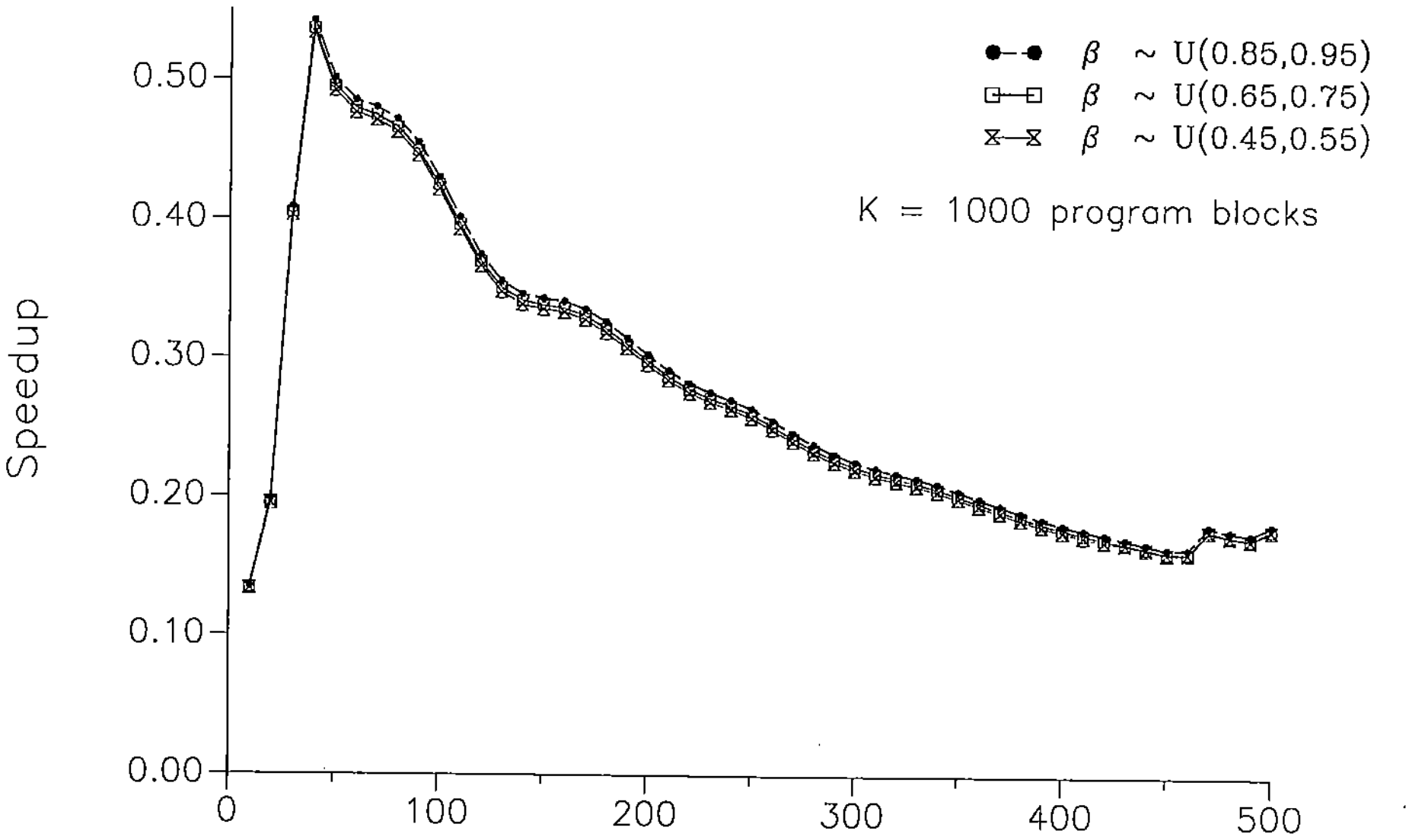
Empirical α_N versus N

Fig. 9



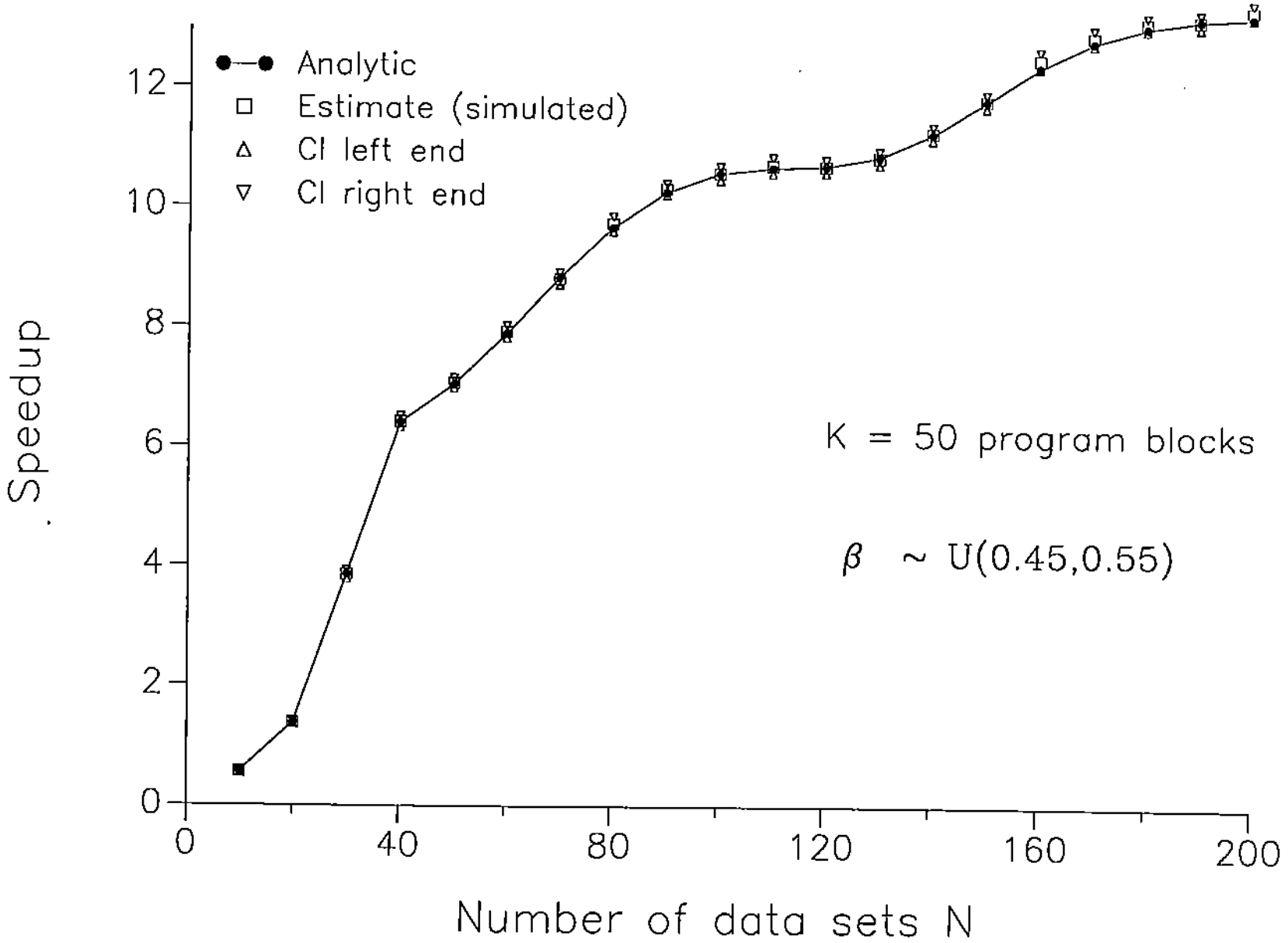
Analytic Speedup for Complete First policy

Fig. 10(a)



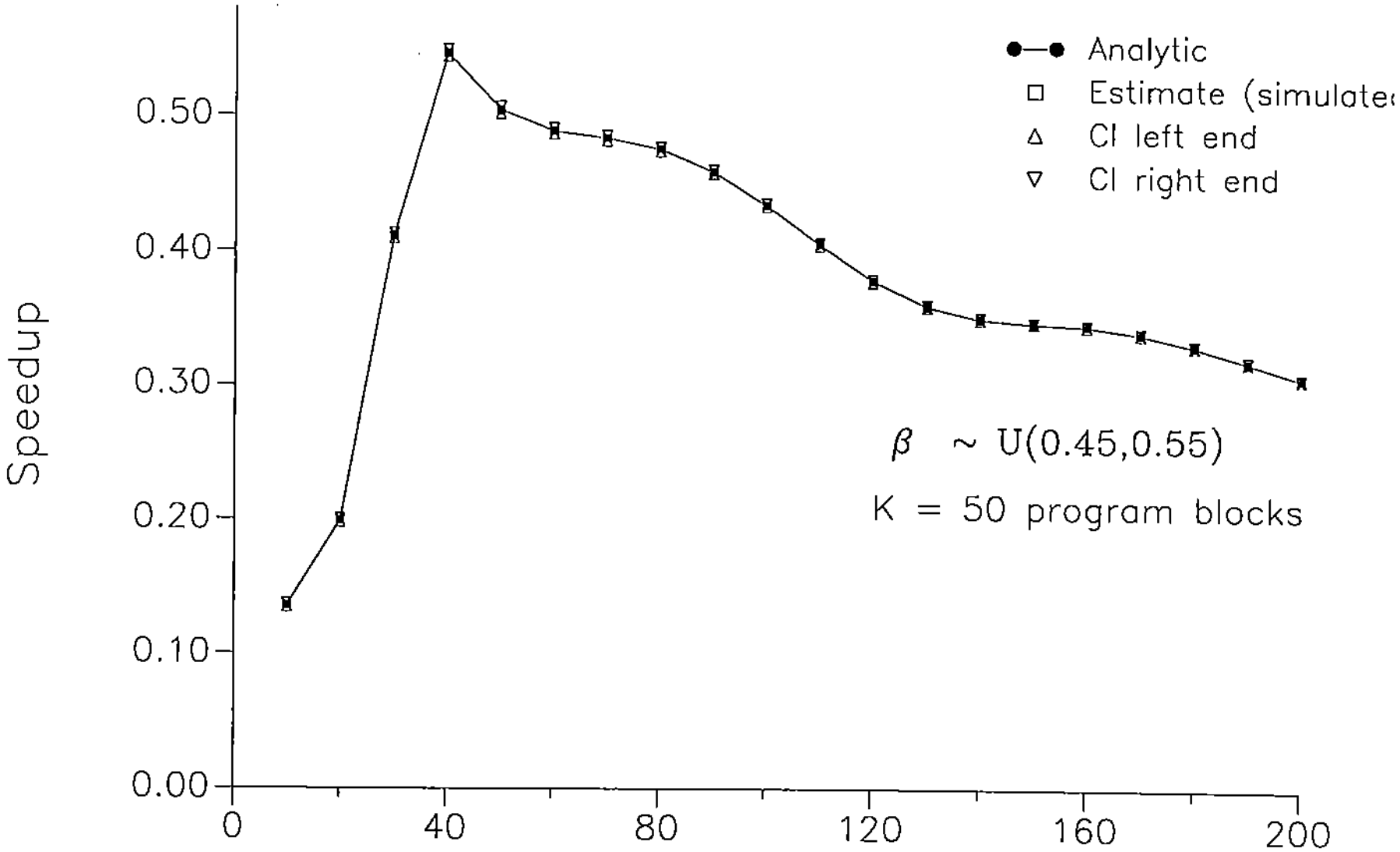
Analytic Speedup for Move Forward policy

Fig. 10(b)



Validation of Simulation: Complete First policy

Fig. 10(c)



Validation of Simulation: Move Forward policy

Fig. 10(d)

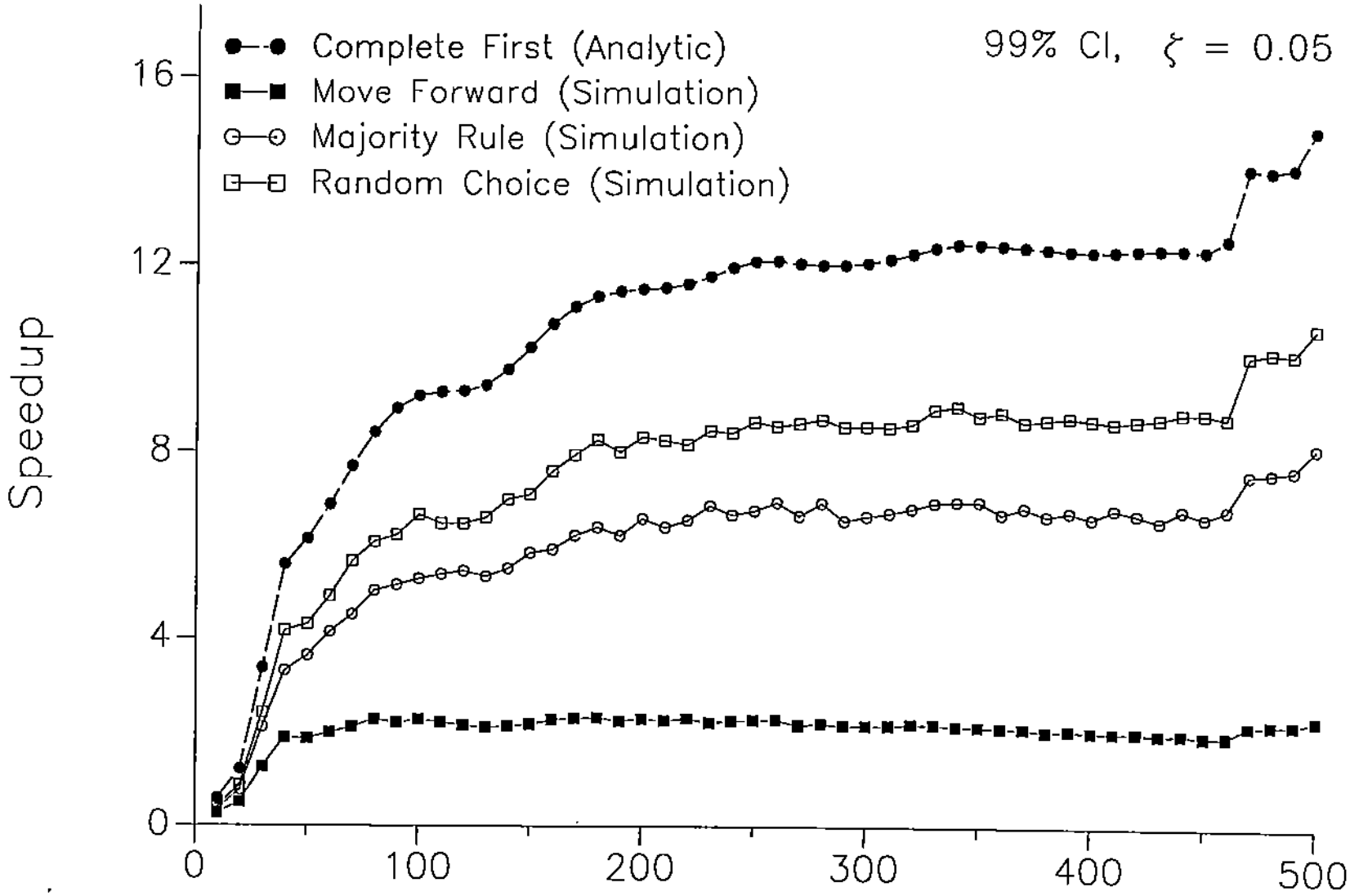
structure. In each case we see that the simulation model closely agrees with the analytic model. In all our simulation experiments, we obtain 99% confidence intervals with relative precision $\zeta = 0.05$, using a students' t-distribution. The relative precision is an upper bound on the ratio of the half-width of the confidence interval at a point to the value of the estimate at that point. The value of ζ that we use ensures a tight confidence interval. A separate validation of the simulation model using the Markov chain based analytic model can be found in [24].

In Fig. 11 we compare the behaviour of $\bar{P}$ for all four policies, using the program graph shown in Fig. 7(d). The curve for the Complete First policy is obtained analytically (see (5.14)), while the rest are obtained through simulation. Once again we see that the Complete First policy outperforms the others, while the Move Forward policy performs poorly. The jump in speedup for N close to 500 is a consequence of an artifact in the spline interpolation (see Fig. 9), as indicated earlier. As discussed prior to the Theorem in section V, the random choice policy outperforms the majority rule policy by being more flexible in its choice of urn selection. For such (nonregressive) graphs, this flexibility results in a behaviour that is closer to the behaviour of the (optimal) Complete First policy.

In Figs. 12(a) and 12(b) we conduct a different kind of experiment, using only simulation, to show that the Complete First policy is not optimal for regressive graphs. For this we use the graph shown in Fig. 7(c). Fig 12(a) shows that the Complete first can be very poor, while the Majority Rule and Random Choice policies can do well when programs have large backward loops. Establishing any kind of optimality here is not a trivial problem. Consider the situation in Fig. 12(b), where we change β_j to 0.999, for all j. Intuitively, we would expect this graph to exhibit behaviour *close* to a nonregressive graph, since backward jumps now occur only with a small probability. Interestingly enough, all four policies appear to do well in this case, with Complete first and Move Forward exhibiting identical behaviour. Such graphs deserve a more detailed analytic investigation, and this we plan to do in our future work.

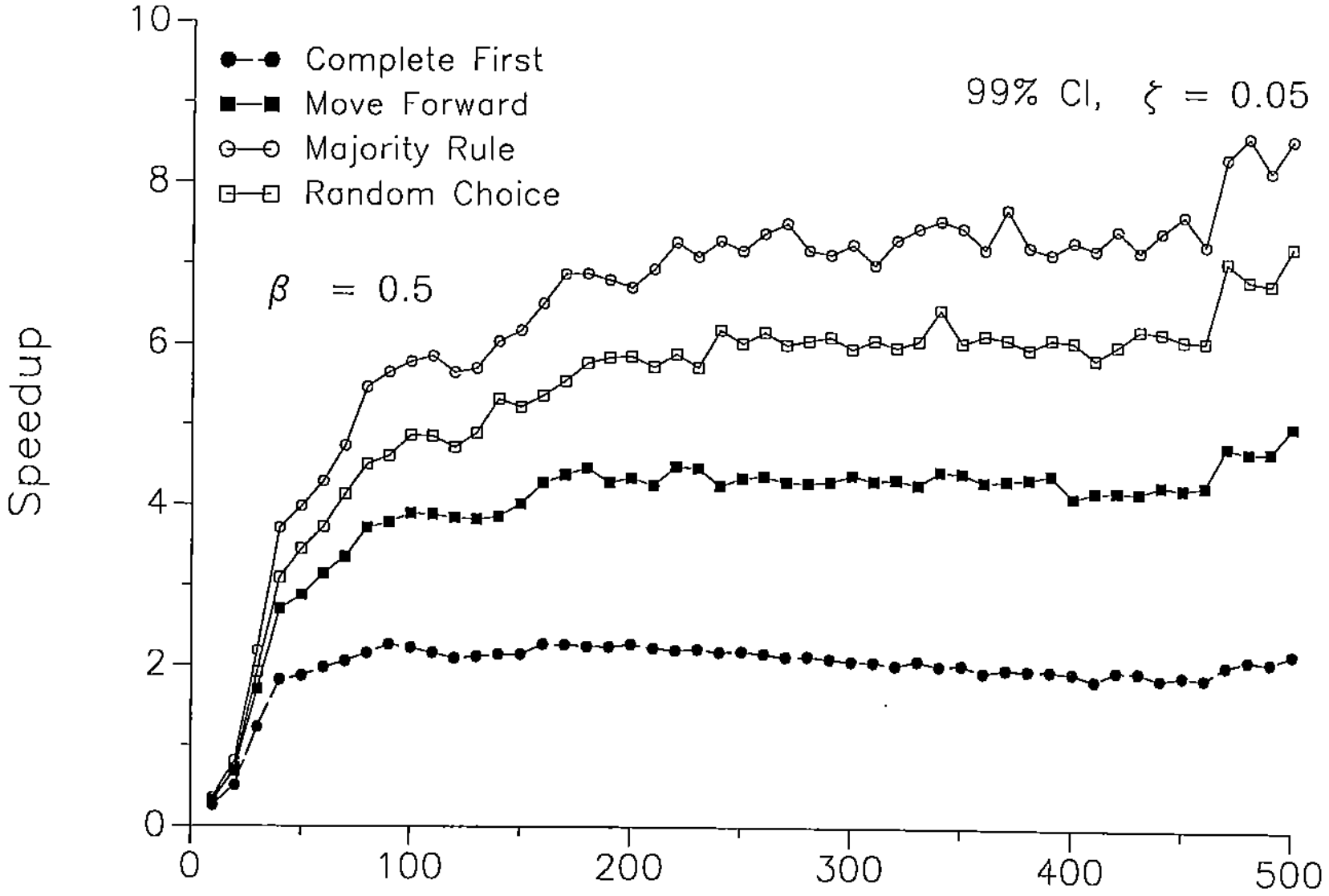
In Figs 13(a) and 13(b) we display simulation results, examining the behaviour of the Complete First and Random Choice policies when up to eight vector processors are available. We use the program graph in Fig. 7(b) with $K = 50$, and each β_j chosen chosen randomly from the interval (0.75,0.85). In both cases we observe that the addition of processors cause an increase in speedup, but only up to a point. Fig. 13(a) displays this fact clearly, showing only a small difference between speedups with four and eight processors, respectively.

In order to understand the effect of vector multiprocessors on $\bar{P}$, we use the analytic model for the Complete First policy (see Fig. 14), with the program graph in Fig. 7(b), and each β_j chosen randomly in the interval (0.85,0.95). If m processors are available, then the N instances of $\bar{P}$ can be equally distributed (assuming that N is a multiple of m) between the processors. If N is not a multiple of m, then the $N - \lfloor N/m \rfloor$ remaining program instances can unified and executed



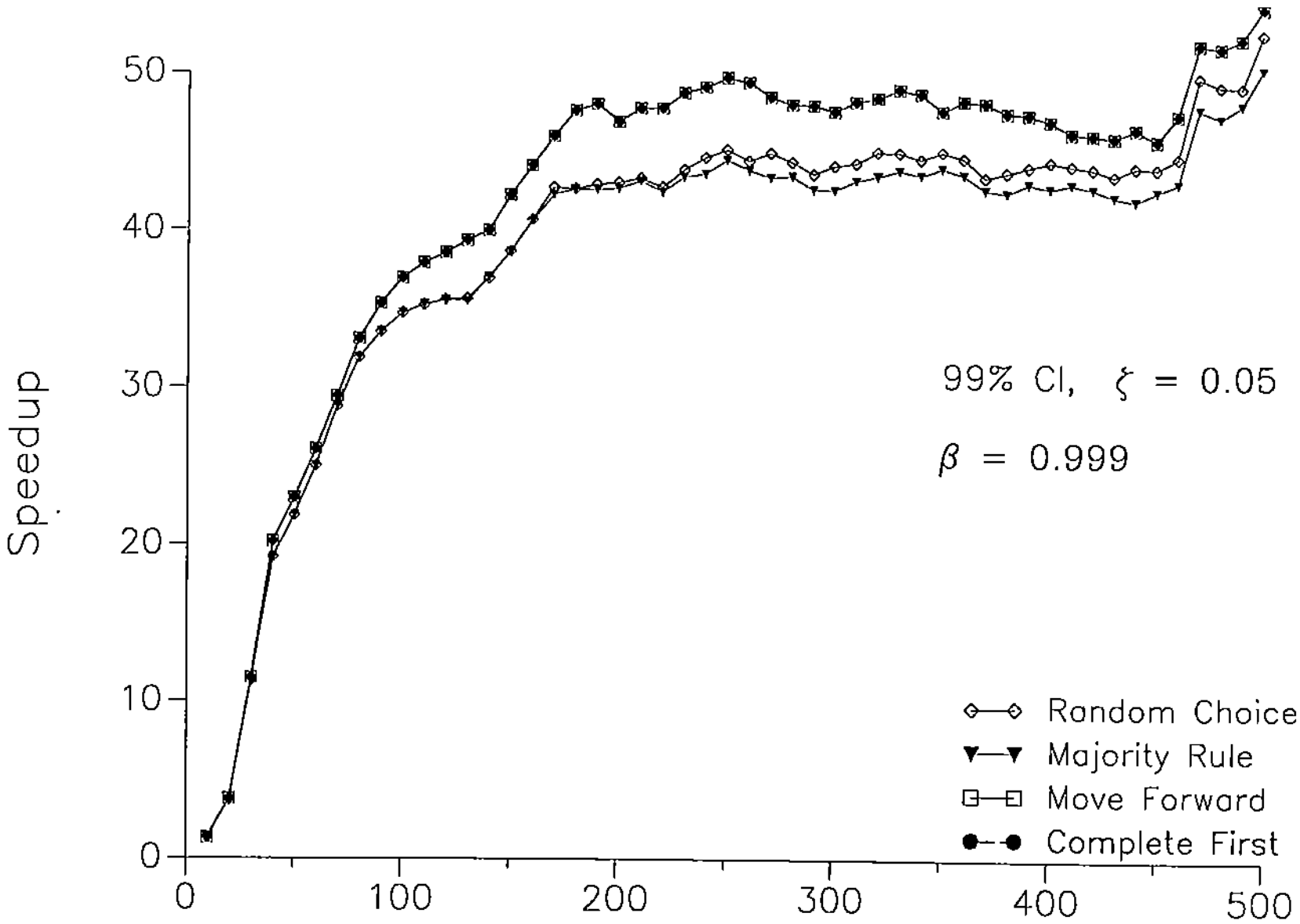
Speedups for various scheduling policies

Fig. 11



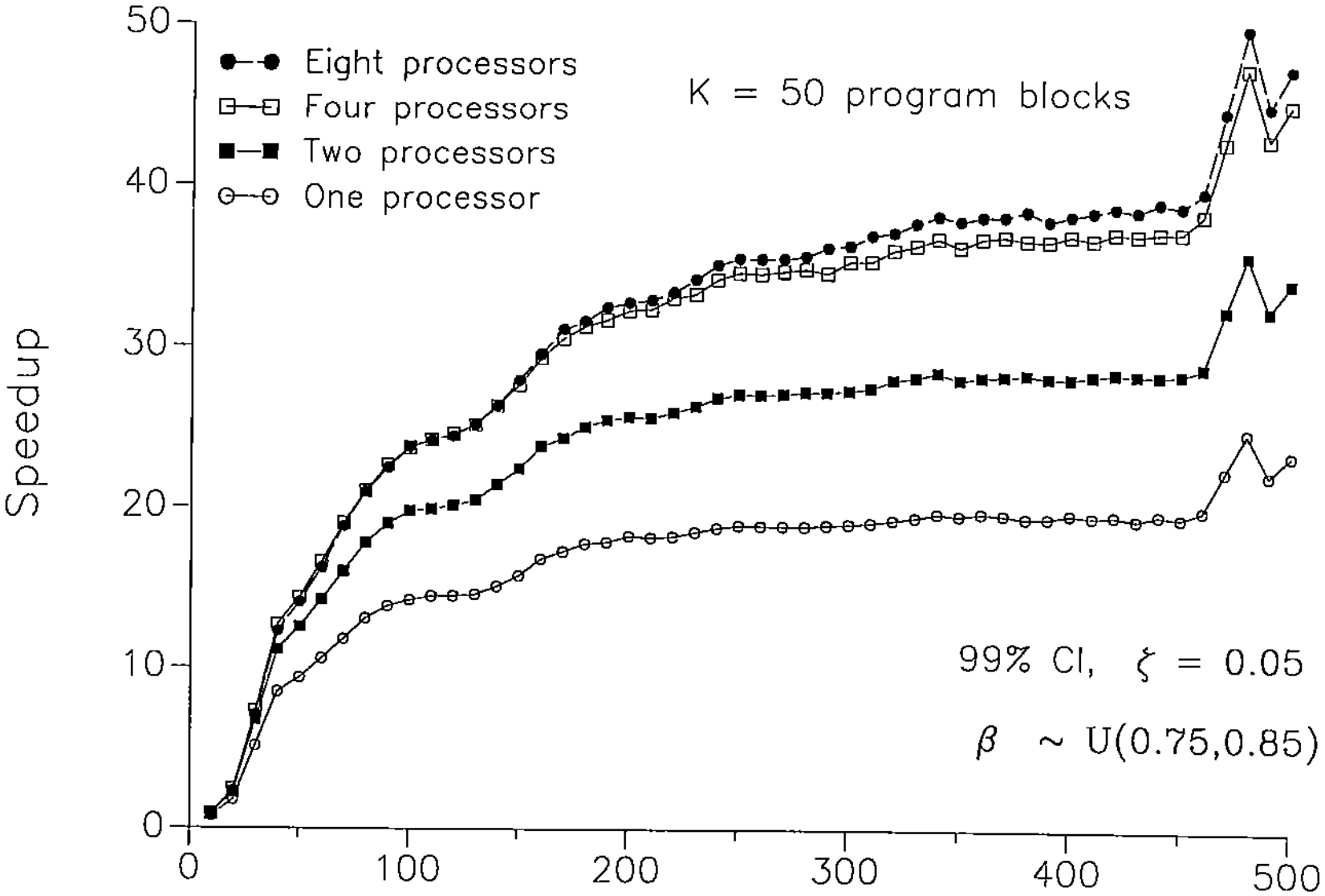
Speedups for a regressive graph (moderate β)

Fig. 12(a)



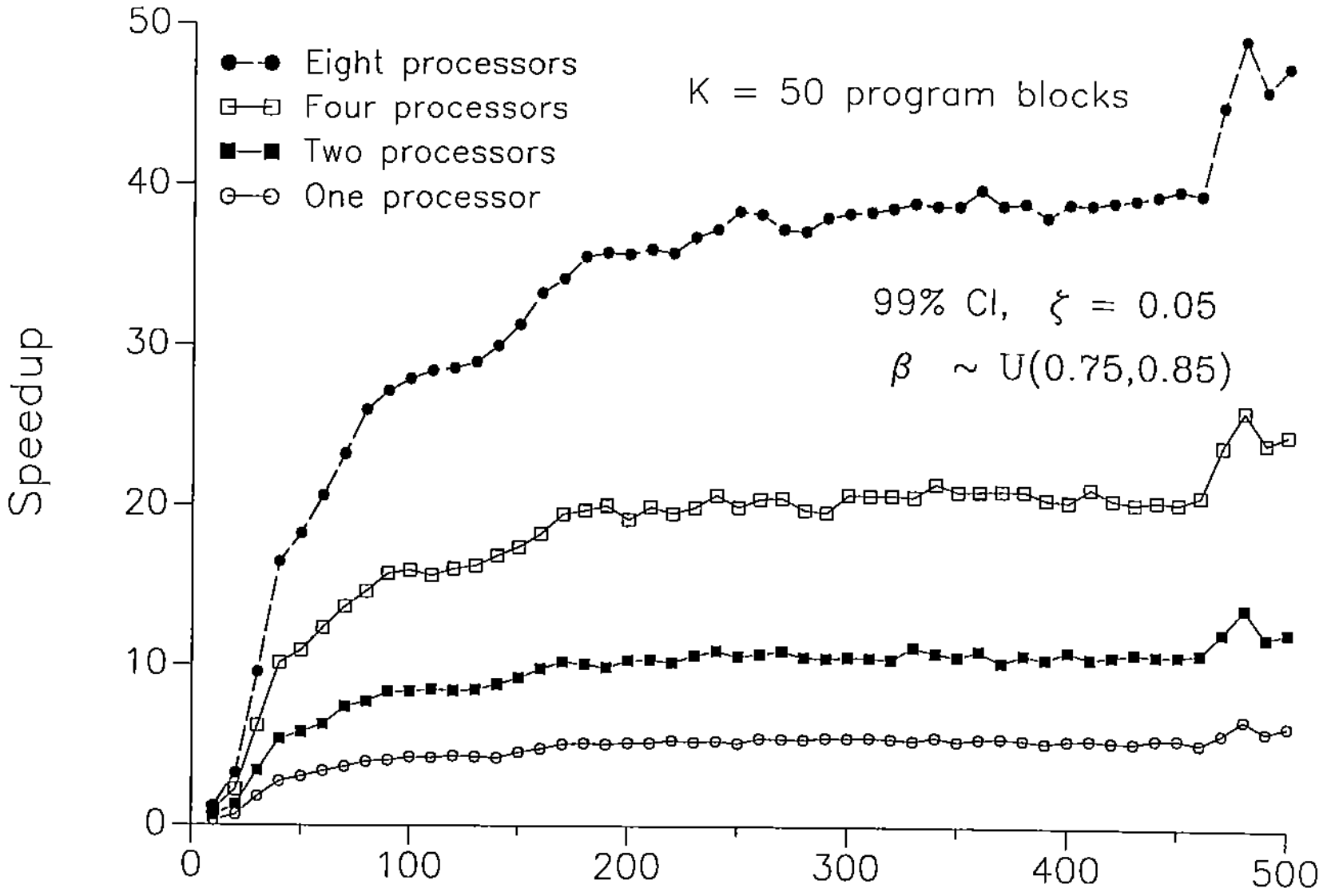
Speedups for a regressive graph (large β)

Fig. 12(b)



Speedup with multiprocessors (Complete First)

Fig. 13(a)



Speedup with multiprocessors (Random Choice)

Fig. 13(b)

after the first batch have completed. However, the dominating portion of $\bar{P}$'s execution time is due to the $\lfloor N/m \rfloor$ instances distributed among the m processors. Fig. 14 displays these speedups for one, two, four and eight processors. In each case, there is always a threshold number of program instances beyond which a system with a greater number of processors will outperform one with a fewer processors. A system with fewer processors will reach its asymptotic speedup value before one with more processors.

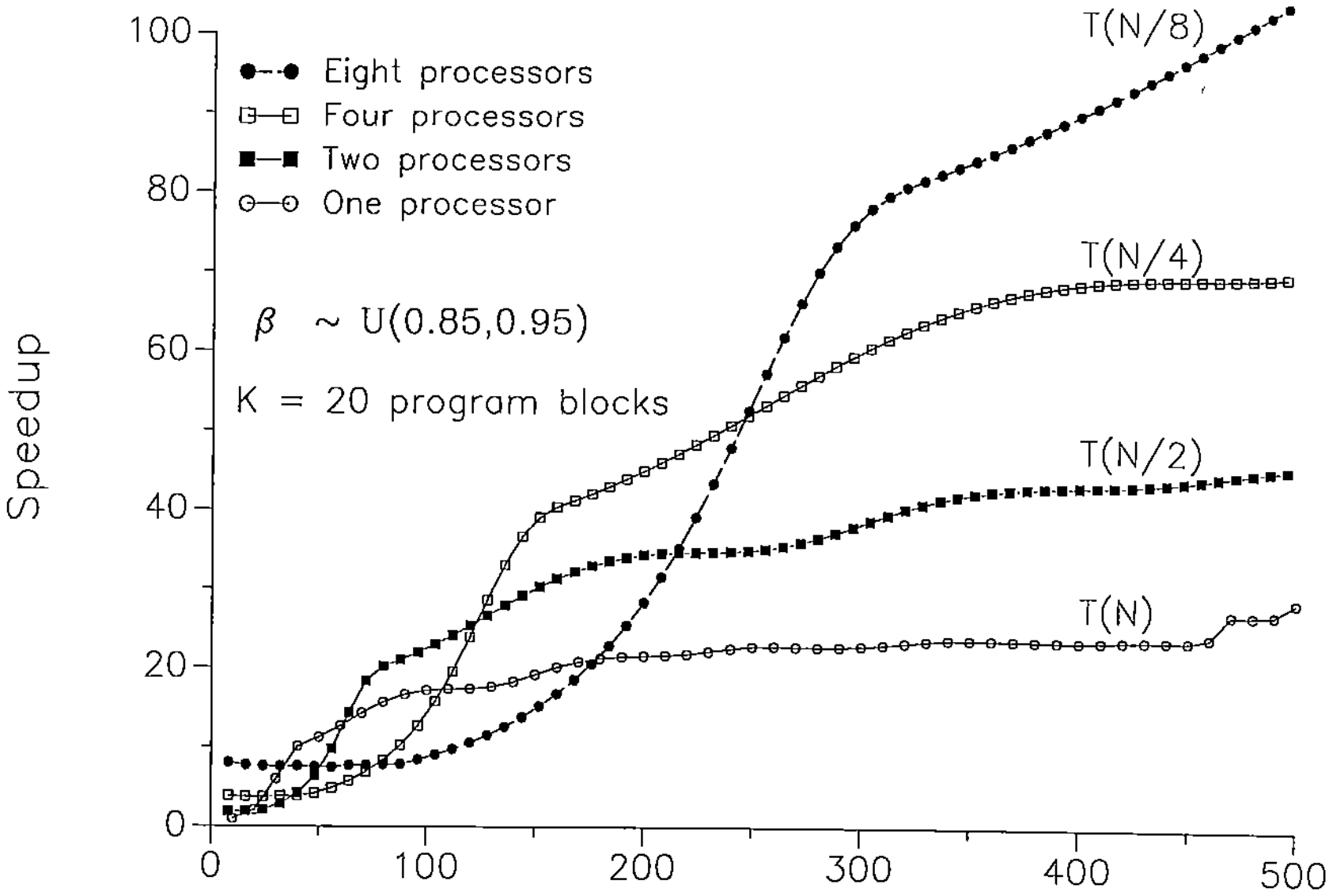
VII. Conclusions and Future Work

In this paper we have demonstrated how program unification could prove to be a very useful technique for improving vectorization and parallelization for a large variety of programs. The analytic model has been used to prove the optimality of the *complete first* policy for block selection for a class of program graphs known as *nonregressive* graphs. Both simulation and analytic results attest to the fact that in many cases, mentioned in section I, unification could lead to speedups of more than 100%.

We are currently working on the development of a tool based on unification. The algorithm presented in section III is the heart of this tool. Such a tool can be used as a front end for a vectorizing compiler or other tools being designed for detecting parallelism for vector-multiprocessors. To cite an application of such a tool, we mention the fact that version 2 of the *MOTHRA* software testing system [1] is being designed with this tool serving as an important component of a transparent interface between the host machine on which the testing tool is implemented and a powerful vector-multiprocessor serving as the backend machine. The unification tool is expected to improve the vector-multiprocessor utilization when a test program or its *mutants* [20] are executed on it for a large set of test cases.

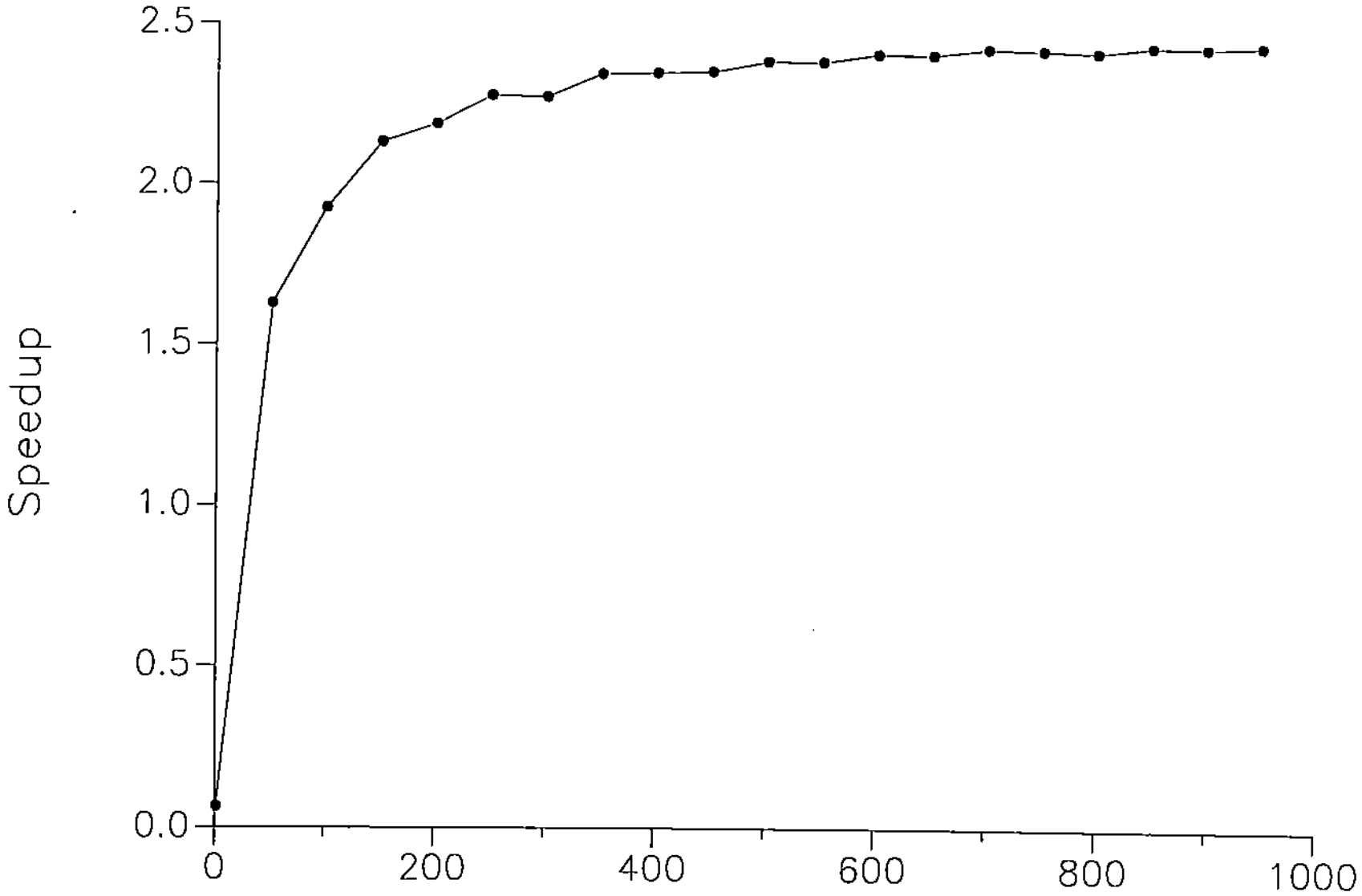
Another related problem on which we are working currently is the reduction of the overhead incurred in managing the multiple paths. The speedup shown in Fig. 6(d) does not account for this overhead. Figure 15(a) shows the speedup when the overhead is taken into account. As is evident from this figure, there is a significant erosion of speedup due to calls to the *outstep* procedure responsible for multiple path management. Figure 15(b) displays the effect of speedup (obtained analytically) for the Complete First policy, using the graph in Fig. 7(b). We take the overhead to be a specified fraction, ranging from 0.0 to 1.6, of the block execution time. Clearly, speedup reduction can be drastic if this fraction is significant.

Besides recoding *outstep* to improve its timing, we are also experimenting with the idea of allocating one of several processors to *outstep*. Thus, the execution of the unified program and *outstep* can be carried out concurrently. However, we have to tackle a problem if we follow this approach. The problem is that we cannot use any of the block selection policies as described in this paper. In order to use any policy *outstep* needs the complete condition vector as computed at



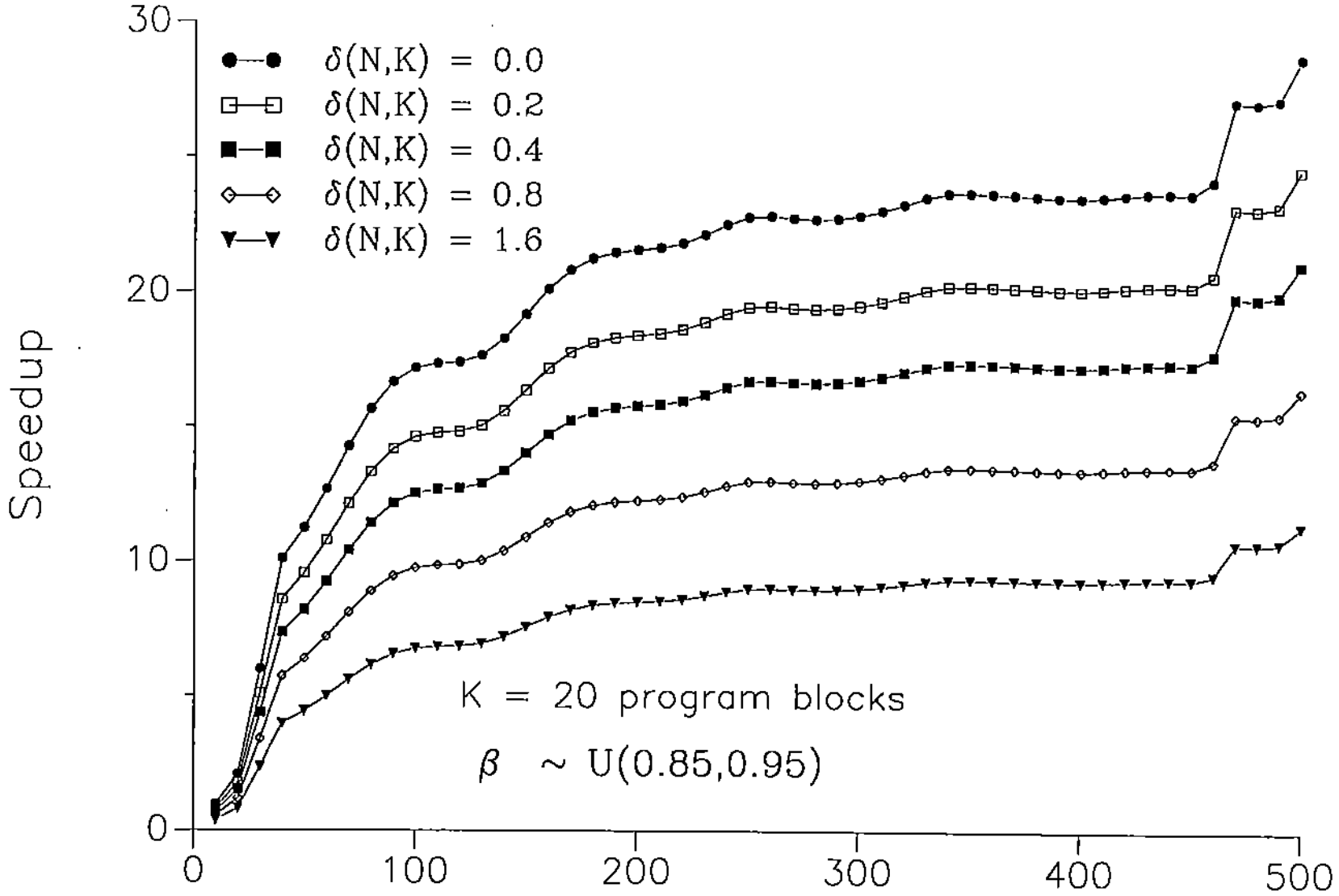
Analytic bounds for vector multiprocessors

Fig. 14



Nonlinear recurrence speedup (including overhead)

Fig. 15(a)



Effect of scheduling overhead on speedup

Fig. 15(b)

the end of a block. If *outstep* is to work in concurrence with the unified program, then it should be able to make a decision about the next block to be executed before the unified program completes the execution of a block. To resolve the dilemma that so arises, we have proposed a *lagged* block selection policy.

The lagged policy implies that *outstep* uses the *previous* condition vector for deciding on the next block to be executed. Certainly this is not expected to be the optimum policy. However, we expect an overall gain in time by the extra concurrency so introduced. This expectation needs to be verified, a task that is currently receiving our attention.

REFERENCES

[1] A.T. Acree and R.A. DeMillo, T.A. Budd and F.G. Sayward, "Mutation Analysis," *Technical Report*, GIT-ICS-79/08, Georgia Institute of technology, Atlanta, GA, 1979.

[2] A.V. Aho, R. Sethi and J.D. Ullman, "Compilers: Principles Techniques and Tools," *Addison-Wesley*, 1986.

[3] H. Duifhuis et al, "Modeling the Cochlear Partition with Coupled Van Der Pol Oscillators," in "Peripheral Auditory Mechanisms," *Lecture Notes in Biomathematics*, No. 64, pp. 290–297, Springer Verlag, Berlin, Aug 1985.

[4] J.R. Allen and K. Kennedy, "A Parallel Programming Environment," *IEEE Software*, pp. 21–29, July 1985.

[5] G.H. Barnes et al, "The ILLIAC IV Computer," *IEEE Trans. on Computers*, pp. 746–757, Aug. 1968.

[6] K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, vol. C29, No. 9, pp. 836–840, 1980.

[7] B. Beizer, *Software System testing and Quality Assurance*, Van Nostrand Reinhold Company, New York, NY, 1984.

[8] V.C. Bhavsar and J.R. Issac, "Design and Analysis of Parallel Monte Carlo Algorithms", *SIAM Journal of Scientific and Statistical Computing*, vol. 8, No. 1, pp 573-595, January 1987.

[9] V.C. Bhavsar and T.A. Tassou, "Monte Carlo Neutron Transport on the Alliant FX/8(Preliminary Results)," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 421–423, 1987.

[10] R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract)," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 836–844, 1986.

[11] Ron Cytron, "Limited Processor Scheduling of Doacross Loops," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 226–234, 1987.

[12] W. Daniel Hillis, "The Connection Machine," *MIT Press*, Cambridge, 1985.

[13] J. Davis et al, "The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 833–835, 1986.

[14] J. P. Hayes et al, "A Microprocessor Based Hypercube Supercomputer," *IEEE Micro*, pp. 6–17, October 1986.

[15] C. Huson et al, "The KAP/205: An Advanced Source-to-Source Vectorizer for the CYBER 205 Supercomputer," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 827–832, 1986.

[16] D.J. Kuck et al, "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance, *Proceedings of Intl. Conf. on Parallel Processing*, pp. 129–138, 1984.

[17] A.P. Mathur and E. Galiano, "Inducing Vectorization: A Formal Analysis," *Technical Report*, SERC-TR-6-P, Software Engineering Research center, Department of Computer Science, Purdue University, W. Lafayette, IN, November 1987 (also to appear in *Proc. of the Third Intl. Conf. On Supercomputing*, Boston, May 1988).

[18] A.P. Mathur, E. Galiano, W. B. Ligon III and T. Greenlaw, "Concurrent Execution Over Multiple Data Sets On Vector Processors," *Technical Report*, SERC-TR-7-P, 1988, Software Engg. Research Center, Dept. of Comp. Sc., Purdue Univ.

[19] A.P. Mathur and E. Galiano, "Concurrent Execution Over Multiple Data Sets On vector Processors, *Technical Report*, SERC-TR-7-P, Software Engineering Research center, Department of Computer Science, Purdue University, W. Lafayette, IN, November 1987.

[20] A.P. Mathur and E.W. Krauser, "Modeling Mutation On A Vector Processor," *to appear in Proceedings of the 10th Intl. Conf. On Software Engineering*, April 11-15, Singapore, 1988.

[21] S.P. Midkiff and D.A. Padua, "Compiler Generated Synchronization For Do Loops," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 544–551, 1986.

[22] S.P. Midkiff and D.A. Padua, "Compiler Algorithms for Synchronization," *IEEE Trans. on Comp.*, Vol. C-36, No. 12, pp. 1485–1495, December 1987.

[23] J. Peir and R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors, *Proceedings Of The Intl. Conf. Parallel Processing*, pp. 217–225, 1987.

[24] V. Rego and A.P Mathur, "Concurrency Enhancement Through Program Unification: A Performance Analysis, *Technical Report*, CSD-TR-739, Department of Computer Science, Purdue University, W. Lafayette, IN, January 1988.

[25] H.J. Siegel et al, "PASM: partitionable SIMD/MIMD system for Image Processing and Pattern Recognition," *IEEE Trans. on Computers*, vol. C30, pp. 934–947, Dec. 1981.

[26] P. Tang and P. Yew, "Processor Self Scheduling for Multiple-Nested Parallel Loops," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 528–535, 1986.

[27] C. Polychronopoulis, "Loop Coalescing: A Compiler Transformation for Parallel Machines, *Proceedings of Intl. Conf. on Parallel Processing*, pp. 235–242, 1987.

[28] C. D. Polychronopoulis et al, "Execution of Parallel Loops on Parallel Processor Systems," *Proceedings of Intl. Conf. on Parallel Processing*, pp. 519–527, 1986.

[29] C. Polychronopoulis et al, "Guided Self Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. on Computers*, vol. C-36, No. 12, pp. 1425–1439, December 1987.

[30] W. Szpankowski and V. Rego, "Yet another application of a binomial recurrence: Order Statistics," Purdue CSD-TR, 1988.

[31] H. Wasserman et al, "A benchmark of the SCS 40 Computer: A Mini Supercomputer Compatible with the Cray XP/24," *Proc. of the vector and Parallel Processors in Computational Science Conference*, Liverpool, England, August 1987.

[32] M. Wolf, "Advanced Loop Interchanging," *Proceedings Of The Intl. Conf. Parallel Processing*, pp. 536–543.

[33] M. Wolf, "Multiprocessor Synchronization for Concurrent Loops," *IEEE Software*, pp. 34–42, January 1988.